Sprawozdanie Struktury danych i złożoność obliczeniowa

Projekt 2

Imię i Nazwisko	Michał Kaźmierczak
Numer indeksu	263924
Termin zajęć	Wtorek, 15^{15}-16^{55} TN
Prowadzący	Mgr. inż. Antoni Sterna

Wprowadzenie

W ramach projektu 2 zadaniem było zaimplementowanie oraz przeprowadzenie pomiarów czasu działania algorytmów grafowych rozwiązujących dwa problemy: wyznaczanie minimalnego drzewa rozpinającego (MST) oraz wyznaczanie najkrótszej ścieżki w grafie. W przypadku MST, zostaną zastosowane dwa algorytmy: algorytm Kruskala oraz algorytm Prima. Natomiast dla problemu najkrótszej ścieżki, wykorzystane zostaną algorytmy Dijkstry oraz Bellmana-Forda.

Implementacja tych algorytmów będzie obejmować dwie reprezentacje grafu w pamięci komputera: reprezentację macierzową, opartą na macierzy wag (macierz sąsiedztwa), oraz reprezentację listową, opartą na połączonych listach sąsiadów.

W ramach tego projektu zostały zaimplementowane wszystkie cztery algorytmy i porównane ich działanie dla dwóch różnych reprezentacji grafu: reprezentacji macierzowej oraz reprezentacji listowej. Pomiar czasu działania algorytmów pozwoli nam na ocenę ich efektywności oraz porównanie wydajności obu reprezentacji grafu.

Czas działania algorytmów zostanie zmierzony na różnych zestawach danych wejściowych, a wyniki pomiarów będą analizowane i przedstawione w dalszej części sprawozdania. W ten sposób będziemy mogli ocenić efektywność poszczególnych algorytmów oraz wpływ wybranej reprezentacji grafu na czas wykonania.

1. Graf macierzy

1.1 Wstęp

Implementacja grafu opiera się na reprezentacji macierzowej, wykorzystując macierz sąsiedztwa do przechowywania wag krawędzi.

Klasa **Edge** reprezentuje krawędź w grafie. Zawiera informacje o wierzchołkach, których krawędź łączy, oraz o jej wadze. Funkcja **compareEdges** jest używana do sortowania krawędzi.

Klasa **GraphMatrix** zawiera logikę związana z grafem w reprezentacji macierzowej. Przechowuje rozmiar grafu, macierz sąsiedztwa **adjacencyMatrix**, wartość **INF** (reprezentującą brak krawędzi), wierzchołek początkowy **vf** i wierzchołek końcowy **vl** dla najkrótszej ścieżki.

Konstruktor **GraphMatrix** inicjalizuje macierz sąsiedztwa **adjacencyMatrix** i wypełnia go wartościami **INF**, tworząc pustą macierz wag.

Metoda **addEdge** dodaje krawędź do grafu. Macierz wypełnia odpowiednią wagą w miejscach **u** oraz **v**. Sprawdza, czy indeksy wierzchołków są prawidłowe. Jeśli graf jest nieskierowany, dodaje również krawędź w drugim kierunku.

Metoda **displayMatrix** wyświetla macierz wag na standardowym wyjściu.

Metoda **findSet** jest pomocniczą funkcją używaną w algorytmie Kruskala do znajdowania zbioru, do którego należy dany wierzchołek.

Metoda **getMinimumSpanningTreeKruskal** implementuje algorytm **Kruskala** do znalezienia minimalnego drzewa rozpinającego.

Metoda **getMinimumSpanningTreePrim** implementuje algorytm **Prima** do znalezienia minimalnego drzewa rozpinającego.

Metoda **shortestPathDijkstra** implementuje algorytm **Dijkstry** do znajdowania najkrótszej ścieżki między wierzchołkiem początkowym a innymi wierzchołkami w grafie.

Metoda **graphMatrixloadFromFile** wczytuje graf z pliku. Pobiera liczbę wierzchołków, liczbę krawędzi, wierzchołek początkowy i końcowy, tworzy nowy graf, wczytuje krawędzie z pliku i dodaje je do grafu. Zwraca wczytany graf.

Fragment implementacji w programie

```
class GraphMatrix {

private:
    int INF = numeric_limits<int>::max(); // Wartość reprezentująca brak krawędzi
    int  num_vertices;
    int **adjacencyMatrix;
    int vf, vl;
public:

    // Konstruktor
    GraphMatrix(int n, int vf, int vl) {
        this-> num_vertices = n; // ilosc wierzcholkow
        this->vf = vf; // wierzcholek poczatkowy dla najkrotszej drogi
        this->vl = vl; // wierzcholek koncowy dla najkrotszej drogi
        fillUpMatrix();
    }

    // Dodaj krawędź o określonej wadze między wierzchołkami u i v
    void addEdge(int u, int v, int weight, bool directed) {
        if (u>num_vertices || v>=num_vertices || weight < 0){
            cout << "index out of bounds" << endl;
            return;
        }
        // zaleznie czy graf jest skierowany
        adjacencyMatrix[u][v] = weight;
        if (!directed){
            adjacencyMatrix[v][u] = weight;
        }
    }
```

2. Graf list

2.1. Wstęp

Implementacja grafu opiera się na reprezentacji listowej.

Klasa **EdgeList** reprezentuje krawędź w grafie. Zawiera informacje o wierzchołkach początkowym i końcowym krawędzi oraz jej wadze. Posiada również statyczną metodę **compareEdges**, która porównuje krawędzie na podstawie ich wag.

Struktura **ListEl** reprezentuje element listy sąsiedztwa. Przechowuje informacje o wierzchołku docelowym krawędzi oraz jej wadze. Posiada wskaźnik **next**, który wskazuje na następny element listy.

Klasa **GraphList** reprezentuje graf w reprezentacji listowej. Zawiera prywatne zmienne, takie jak liczba wierzchołków (**num_vertices**), wierzchołek początkowy (**vf**) i wierzchołek końcowy (**vl**). Przechowuje listę wskaźników na elementy **ListEl** w tablicy **adj_list**.

Konstruktor klasy **GraphList** inicjalizuje tablicę **adj_list** o rozmiarze **num_vertices** i ustawia wskaźniki na **nullptr**.

Metoda **addEdge** dodaje krawędź do grafu. Sprawdza, czy indeksy wierzchołków są prawidłowe, a następnie tworzy nowy element **ListEl** i dodaje go do listy sąsiedztwa odpowiedniego wierzchołka. Jeśli graf jest nieskierowany, dodaje również krawędź w drugim kierunku.

Metoda **displayList** wyświetla listę sąsiedztwa dla każdego wierzchołka grafu.

Metoda **findSet** jest pomocniczą funkcją używaną w algorytmie Kruskala do znajdowania zbioru, do którego należy dany wierzchołek.

Metoda **getMinimumSpanningTreeKruskal** implementuje algorytm Kruskala do znajdowania minimalnego drzewa spinającego (MST)

Metoda **getMinimumSpanningTreePrim** implementuje algorytm Prima do znajdowania minimalnego drzewa spinającego (MST)

Metoda **shortestPathDijkstra** implementuje algorytm Dijkstry do znajdowania najkrótszych ścieżek w grafie.

Metoda **graphListloadFromFile** wczytuje graf z pliku. Pobiera liczbę wierzchołków, liczbę krawędzi, wierzchołek początkowy i końcowy, tworzy nowy graf, wczytuje krawędzie z pliku i dodaje je do grafu. Zwraca wczytany graf.

Fragment implementacji w programie

```
class GraphMatrix {

private:
    int INF = numeric_limits<int>::max(); // Wartość reprezentująca brak krawędzi
    int  num_vertices;
    int **adjacencyMatrix;
    int vf, vl;
public:

    // Konstruktor
    GraphMatrix(int n, int vf, int vl) {
        this-> num_vertices = n; // ilosc wierzcholkow
        this->vf = vf; // wierzcholek poczatkowy dla najkrotszej drogi
        this->vl = vl; // wierzcholek koncowy dla najkrotszej drogi
        fillUpMatrix();
    }

    // Dodaj krawędź o określonej wadze między wierzchołkami u i v
    void addEdge(int u, int v, int weight, bool directed) {
        if (u>num_vertices || v>=num_vertices || weight < 0){
            cout << "index out of bounds" << endl;
            return;
        }
        // zaleznie czy graf jest skierowany
        adjacencyMatrix[u][v] = weight;
        if (!directed){
            adjacencyMatrix[v][u] = weight;
        }
    }
```

3. Złożoności obliczeniowe algorytmów.

Badania zostały przeprowadzone dla pięciu różnych reprezentatywnych wartości liczby wierzchołków (10, 50, 100, 250, 500). Dla każdej liczby wierzchołków przeprowadzono pomiary dla czterech różnych gęstości grafu: 25%, 50%, 75% i 99%.

W celu uzyskania wyników uśrednionych, dla każdego zestawu reprezentacji grafu, liczby wierzchołków i gęstości, wykonano 100 losowych instancji. Podczas pomiarów zarejestrowano czas działania algorytmów dla każdej instancji, a następnie obliczono średni czas dla danego zestawu. Pozwoliło to na porównanie wydajności algorytmów w różnych warunkach i na ocenę, jak zmiany w rozmiarze grafu oraz jego gęstości wpływają na czas wykonania.

Generowanie instancji

```
static vector<GraphList> generateGraphInstances(int numInstances, int numVertices, double density) {
    random_device rd; // obiekt do tworzenia liczb losowych
    mt19937 gen( sd: rd()); // generowanie liczb pseudolosowych
    uniform_real_distribution<double> dis( a: 0.0,  b: 1.0); // tworzy obiekt rozkładu jednorodnego

    vector<GraphList> instances;

    for (int i = 0; i < numInstances; ++i) {
        GraphList graph( num_vertices: numVertices,  vf: 0,  vl: 0);
        for (int u = 0; u < numVertices; ++u) {
            for (int v = u + 1; v < numVertices; ++v) {
                if (dis( &: gen) <= density) {
                    int weight = rand() % 10 + 1;  // Losowa waga z zakresu 1-10
                    graph.addEdge(u, v, weight,  directed: false);  // Dodanie nieskierowanej krawędzi
                }
            }
        }

        instances.push_back(graph);
    }

    return instances;
}
```

Funkcja rozpoczyna od utworzenia obiektu **random_device**, który służy do generowania liczb losowych. Następnie, przy użyciu generatora liczb pseudolosowych **mt19937** i rozkładu jednorodnego **uniform_real_distribution**.

Następnie, dla każdej pary wierzchołków **u** i **v**, gdzie **u** jest mniejsze niż **v**, sprawdzane jest, czy losowa wartość generowana przez **dis(gen)** jest mniejsza lub równa zadanej gęstości **density**. Jeśli tak, to tworzona jest losowa waga krawędzi z przedziału od 1 do 10 (przy użyciu **rand() % 10 + 1)** i dodawana jest nieskierowana krawędź pomiędzy wierzchołkami u i v do grafu za pomocą metody addEdge obiektu graph.

3.1. Algorytmy MST.

3.1.1. Algorytm Kruskala graf macierzy

Złożoność obliczeniowa funkcji **getMinimumSpanningTreeKruskal** dla implementacji grafu w postaci grafu macierzy, w zależności od liczby wierzchołków (**V**) i krawędzi (**E**) w grafie wynosi:

Tworzenie listy krawędzi: **O(V^2)** - dwie zagnieżdżone pętle iterujące po wierzchołkach i tworzące krawędzie.

Sortowanie krawędzi: **O(E log E)** - gdzie **E** to liczba krawędzi. Sortowanie krawędzi rosnąco według wag. (Z dokumentacji algorytmu)

Inicjalizacja zbiorów rozłącznych: **O(V)** - inicjalizacja wektora grup.

Przechodzenie po posortowanych krawędziach: **O(E)** - iteracja po wszystkich krawędziach.

W sumie złożoność obliczeniowa funkcji **getMinimumSpanningTreeKruskal** wynosi **O(V^2 log V)** w najgorszym przypadku, gdzie V to liczba wierzchołków w grafie. W praktyce, gdy **E** jest bliskie **V^2**, złożoność wynosi **O(E log V).**

gęstość\v	10	50	100	250	500
25%	5,1709	31,869	51,823	127,37	262,07
50%	6,139	25,032	51,061	137,03	263,78
75%	5,1045	27,86	51,399	144,27	278,13
99%	4,7927	23,399	69,374	135,37	288,18

3.1.2. Algorytm Prima graf Macierzy

Złożoność obliczeniowa funkcji **getMinimumSpanningTreePrim** dla implementacji grafu w postaci grafu macierzy w zależności od liczby wierzchołków (**V**) i krawędzi (**E**) w grafie wynosi:

Inicjalizacja tablic i kolejki priorytetowej: **O(V).**

Przeszukiwanie grafu: **O(V)** razy (dla każdego wierzchołka).

Wewnętrzna pętla przeglądająca sąsiadujące wierzchołki: **O(E)** razy (dla każdej krawędzi).

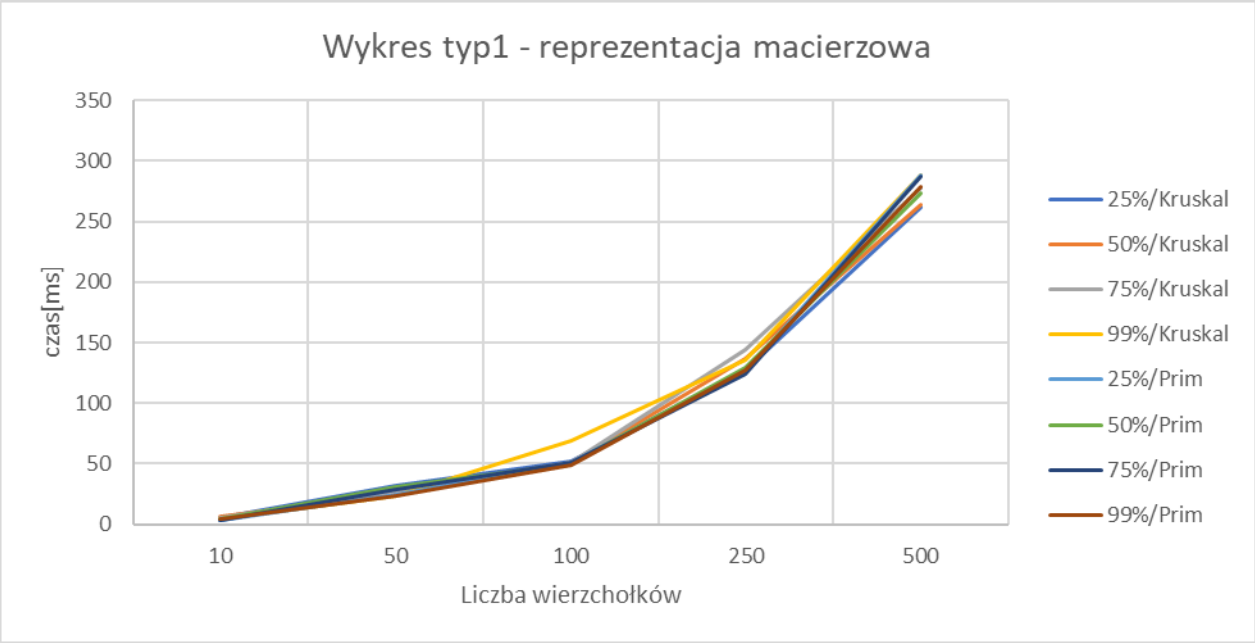
Wstawianie i pobieranie elementów z kolejki priorytetowej: **O((V + E) log V)** w sumie (wstawienie i pobranie dla każdego wierzchołka i krawędzi).

Dodawanie krawędzi do MST: **O(V)** razy (dla każdego wierzchołka).

Ostatecznie, złożoność obliczeniowa wynosi O((V + E) log V). Głównym czynnikiem wpływającym na złożoność jest operacja wstawiania i pobierania elementów z kolejki priorytetowej, która ma złożoność logarytmiczną.

gęstość\v	10	50	100	250	500
25%	3,1481	24,385	51,335	126,75	288,41
50%	3,9453	31,024	49,642	129,81	273,25
75%	3,6409	28,79	50,96	124,46	287,61
99%	4,2654	23,918	49,12	126,81	279,27

3.1.3. Wykres Typ1.



Na wykresie nie widać jednoznacznie różnicy między wydajnością algorytmów, może być to spowodowane zbyt małą ilością wierzchołków, lub samą specyfikacją sprzętu oraz kompilatora.

3.1.4 Algorytm Kruskala graf listy sąsiadów

Złożoność obliczeniowa funkcji **getMinimumSpanningTreeKruskal** dla implementacji grafu w postaci grafu listy sąsiadów, w zależności od liczby wierzchołków (**V**) i krawędzi (**E**) w grafie wynosi:

Tworzenie listy krawędzi: **O(E)**, gdzie **E** to liczba krawędzi w grafie. W tym przypadku, iteracja po liście sąsiedztwa i tworzenie krawędzi.

Sortowanie krawędzi: **O(E log E),** gdzie **E** to liczba krawędzi. Sortowanie krawędzi rosnąco według wag. (Z dokumentacji algorytmu)

Inicjalizacja zbiorów rozłącznych: **O(V)**, gdzie **V** to liczba wierzchołków. Inicjalizacja wektora grup.

Przechodzenie po posortowanych krawędziach: **O(E),** gdzie **E** to liczba krawędzi. Iteracja po wszystkich krawędziach.

Ostateczna złożoność obliczeniowa funkcji **getMinimumSpanningTreeKruskal** wynosi **O(E log E),** gdzie **E** to liczba krawędzi w grafie. Złożoność ta wynika głównie z sortowania krawędzi.

gęstość\v	10	50	100	250	500
25%	3,2291	23,114	48,443	121,94	259,55
50%	3,5046	22,45	46,918	129,33	297,41
75%	4,0785	24,099	49,436	134,32	311,06
99%	3,4937	22,465	49,102	140,78	392,66

3.1.5 Algorytm Prima graf listy sąsiadów

Złożoność obliczeniowa funkcji **getMinimumSpanningTreePrim** można oszacować na **O((V + E) log V),** gdzie **V** to liczba wierzchołków, a **E** to liczba krawędzi w grafie.

Inicjalizacja tablic i kolejki priorytetowej: **O(V).**

Przeszukiwanie grafu: **O(V)** razy (dla każdego wierzchołka).

Wewnętrzna pętla przeglądająca sąsiadujące wierzchołki: **O(E)** razy (dla każdej krawędzi).

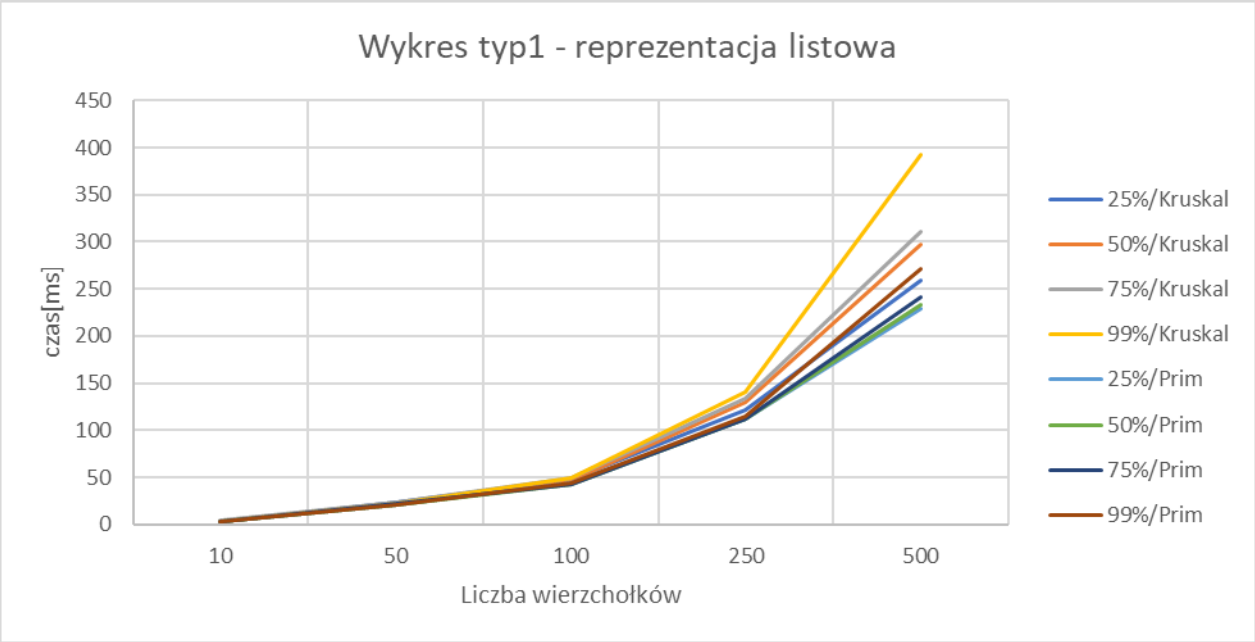
Wstawianie i pobieranie elementów z kolejki priorytetowej: **O((V + E) log V)** w sumie (wstawienie i pobranie dla każdego wierzchołka i krawędzi).

Dodawanie krawędzi do MST: **O(V)** razy (dla każdego wierzchołka).

Ostatecznie, złożoność obliczeniowa wynosi **O((V + E) log V)**, gdzie **V** to liczba wierzchołków, a **E** to liczba krawędzi w grafie. Głównym czynnikiem wpływającym na złożoność jest operacja wstawiania i pobierania elementów z kolejki priorytetowej, która ma złożoność logarytmiczną.

gęstość\v	10	50	100	250	500
25%	3,1655	20,701	43,553	112,06	228,99
50%	3,3361	20,842	43,079	111,73	233,28
75%	3,3167	22,422	42,779	111,4	241,12
99%	3,5491	20,551	43,23	114,37	271,06

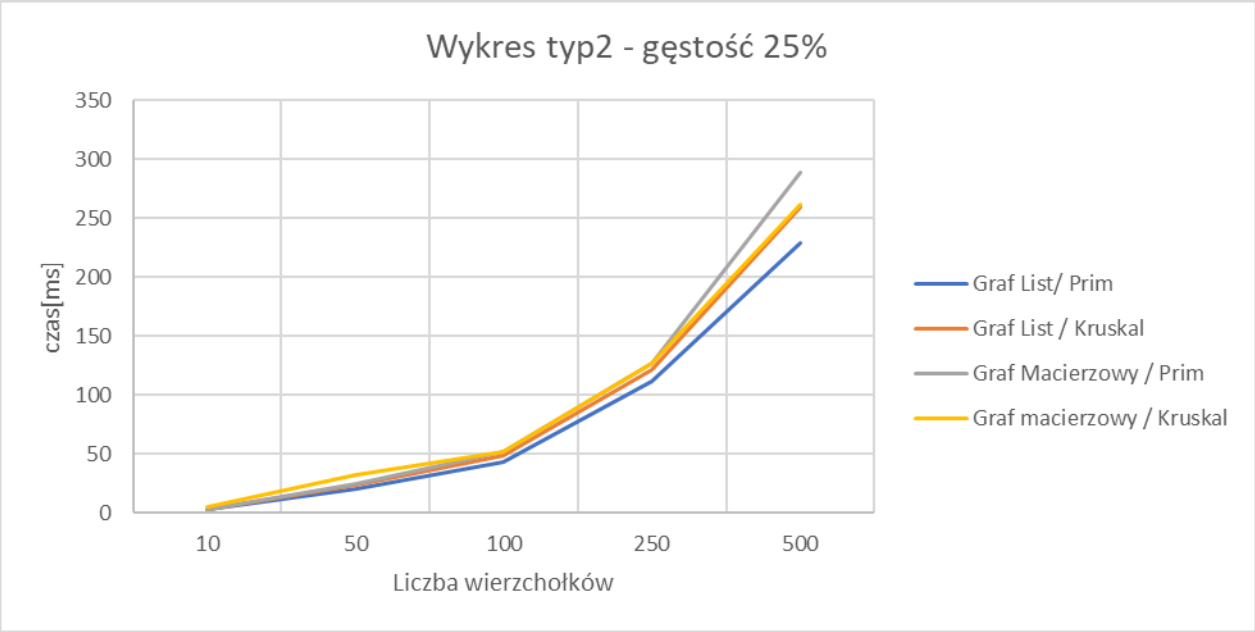
3.1.6 Wykres Typ1

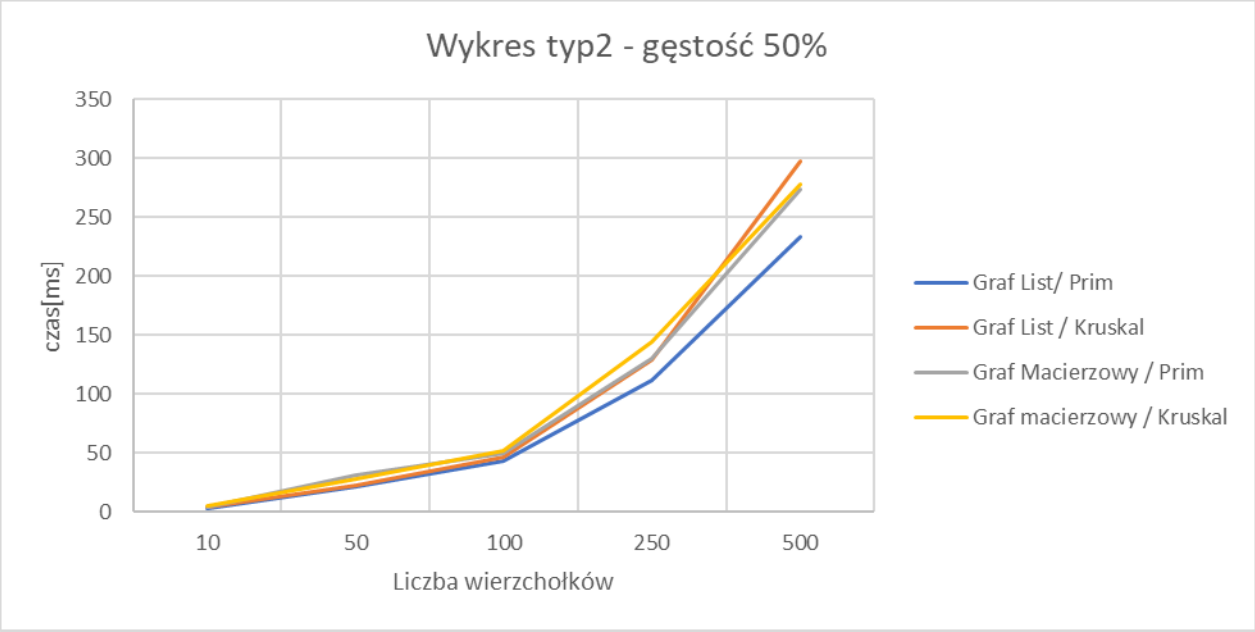


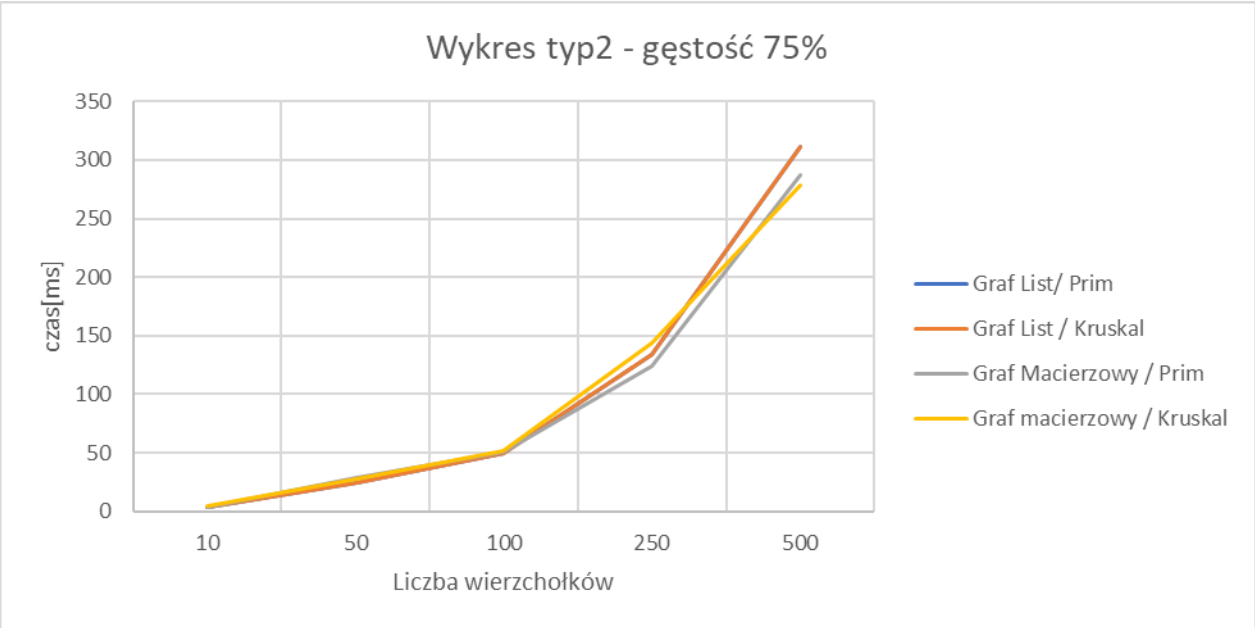
Na tym wykresie widać znaczącą przewagę algorytmu Prima nad algorytmem Kruskala. Może być to spowodowane wykorzystaniem listy sąsiadów, która jest szybsza w obsłudze i daje przewagę nad grafem w reprezentacji macierzowej.

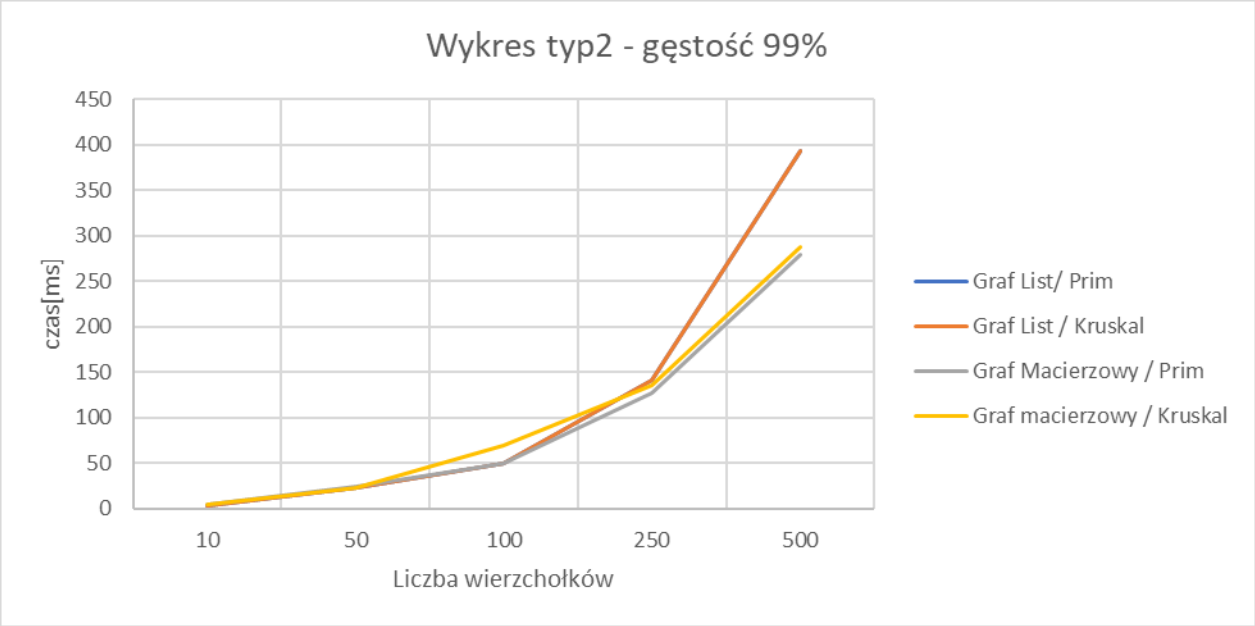
3.1.7 Podsumowanie

Wykresy typ2. parametryzowane typem algorytmu i typem reprezentacji grafu









Dla niższych gęstości przewagę ma Graf reprezentacji list sąsiadów jednakże, gęstości powyżej 50% przewaga jest po stronie grafu macierzy. Jest to spowodowane tym, że dla mniejszej ilości krawędzi iterowanie po liście jest wydajniejsze niż iteracja po macierzy.

3.2. Algorytmy najkrótszej ścieżki w grafie.

3.2.1. Algorytm Dijkstry graf listy sąsiadów

Złożoność obliczeniowa funkcji **shortestPathDijkstra**() wynosi **O(V^2),** gdzie **V** to liczba wierzchołków, a **E** to liczba krawędzi w grafie.

Algorytm Dijkstry polega na iteracyjnym znajdowaniu wierzchołka o najmniejszej odległości i aktualizowaniu odległości do sąsiadujących wierzchołków. W najgorszym przypadku dla każdego wierzchołka musimy przeglądać wszystkie pozostałe wierzchołki w celu znalezienia wierzchołka o najmniejszej odległości.

W funkcji znajduje się pętla główna, która iteruje V razy (dla każdego wierzchołka) i pętla wewnętrzna, która również iteruje V razy (dla każdego wierzchołka). Co daje **O(V^2).**

Następnie, dla każdej krawędzi wychodzącej z aktualnego wierzchołka, sprawdzamy, czy można poprawić odległość do tego wierzchołka. W najgorszym przypadku, każda krawędź będzie musiała zostać zaktualizowana raz. Ponieważ w grafie może być maksymalnie **E** krawędzi. Gdzie w najgorszym przypadku **E=V-1**.

Ostatecznie, całkowita złożoność obliczeniowa wynosi **O((V^2),** ponieważ dla każdego wierzchołka musimy sprawdzić wszystkie incydentne krawędzie, a całość powtarzamy **V** razy (dla każdego wierzchołka).

gęstość\v	10	50	100	250	500
25%	0,369	0,5756	0,7209	2,3579	8,7719
50%	0,4153	0,4646	0,6861	2,8699	10,3
75%	0,3973	0,477	0,704	2,9994	11,912
99%	0,406	0,5186	0,8437	4,0471	14,335

3.2.2. Algorytm Bellmana-Forda graf listy sąsiadów

Złożoność funkcji **shortestPathbellmanFord**() wynosi **O(V^2 * E),** gdzie **V** oznacza liczbę wierzchołków, a **E** oznacza liczbę krawędzi w grafie.

W algorytmie **Bellmana-Forda** iterujemy V - 1 razy, relaksując wszystkie krawędzie w każdej iteracji.

Relaksacja krawędzi polega na porównaniu aktualnej odległości wierzchołka docelowego z odległością do wierzchołka początkowego plus wagą krawędzi. Jeśli aktualna odległość jest większa, zostaje zaktualizowana.

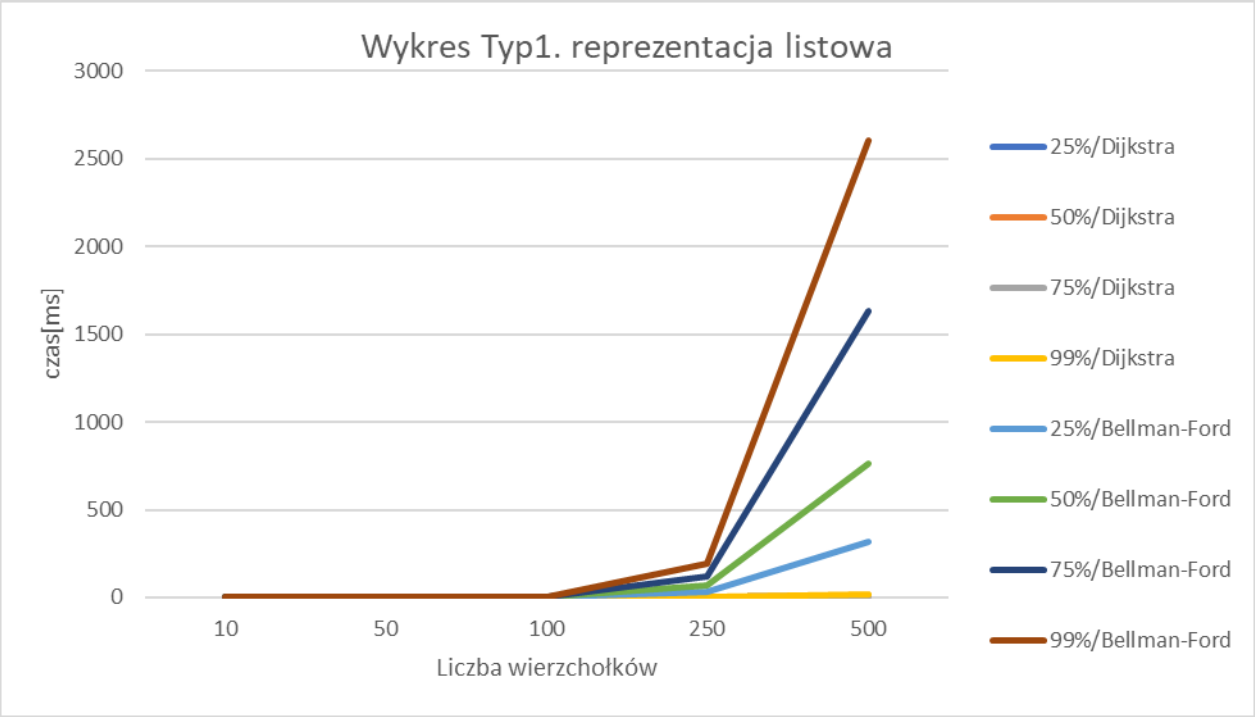
Ponieważ każdą krawędź relaksujemy co najwyżej **V - 1** razy, a pętla wewnętrzna iteruje V razy to złożoność czasowa wynosi **O(V(V - 1) * E),** co można uprościć do **O(V^2* E).**

Następnie sprawdzane jest wystąpienie ujemnych cykli, gdzie ponownie iterujemy po wszystkich krawędziach i sprawdzamy, czy istnieje krawędź, dla której odległość do wierzchołka docelowego może być skrócona. Jeśli tak, to oznacza, że graf zawiera ujemny cykl.

Złożoność tej części kodu wynosi również **O(V^2 * E),** ponieważ iterujemy po wszystkich krawędziach i dla każdej krawędzi wykonujemy stałą liczbę operacji.

gęstość\v	10	50	100	250	500
25%	0,003	0,2373	1,6815	31,118	314,77
50%	0,0052	0,4077	3,2222	71,897	765,89
75%	0,0059	0,6072	4,8862	118,84	1635,7
99%	0,0072	0,7827	6,8044	191,97	2604,5

3.2.3. Wykres Typ1.



Na wykresie widać znaczącą przewagę algorytmu Dijkstra nad algorytmem Bellmana-Forda. Jednakże algorytm Bellmana-Forda potrafi rozpatrywać cykle ujemne, przez co może być wykorzystany w specyficznych sytuacjach dla grafów z ujemnymi wagami. Ponadto gęstości dla obu algorytmów wpływają na jego wydajność. **O(V^2) < O(V^2 * E).** Warto również zauważyć, że algorytm Dijkstra zdobywa znaczącą przewagę dopiero dla większej ilości wierzchołków.

3.2.4. Algorytm Dijkstry graf macierzy

Funkcja **shortestPathDijkstra()** ma złożoność obliczeniową **O(V^2),** gdzie **V** oznacza liczbę wierzchołków w grafie.

Algorytm Dijkstry polega na iteracyjnym znajdowaniu wierzchołka o najmniejszej odległości i aktualizowaniu odległości do sąsiadujących wierzchołków. W najgorszym przypadku dla każdego wierzchołka musimy przeglądać wszystkie pozostałe wierzchołki w celu znalezienia wierzchołka o najmniejszej odległości.

Pierwsza pętla for wykonuje się **V** razy, ponieważ iteruje po wszystkich wierzchołkach.

W każdej iteracji znajduje wierzchołek o najmniejszej odległości, co zajmuje **O(V)** czasu.

Druga pętla for również wykonuje się **V** razy, ponieważ dla każdego wierzchołka aktualizuje odległości do jego sąsiadujących wierzchołków.

W najgorszym przypadku musimy sprawdzić odległość dla wszystkich par wierzchołków, co zajmuje **O(V)** czasu.

Zatem łączna złożoność obliczeniowa wynosi **O(V^2).**

gęstość\v	10	50	100	250	500
25%	0,004	0,0887	0,3448	2,3724	7,7973
50%	0,0058	0,1003	0,3967	2,3068	8,5384
75%	0,0045	0,0939	0,3607	2,0972	7,8565
99%	0,0042	0,0864	0,3488	1,9144	7,59

3.2.5. Algorytm Bellmana-Forda graf macierzy.

Złożoność funkcji **shortestPathbellmanFord**() wynosi **O(V^3),** gdzie **V** oznacza liczbę wierzchołków..

W algorytmie **Bellmana-Forda** iterujemy V - 1 razy, relaksując wszystkie krawędzie w każdej iteracji.

Relaksacja krawędzi polega na porównaniu aktualnej odległości wierzchołka docelowego z odległością do wierzchołka początkowego plus wagą krawędzi. Jeśli aktualna odległość jest większa, zostaje zaktualizowana.

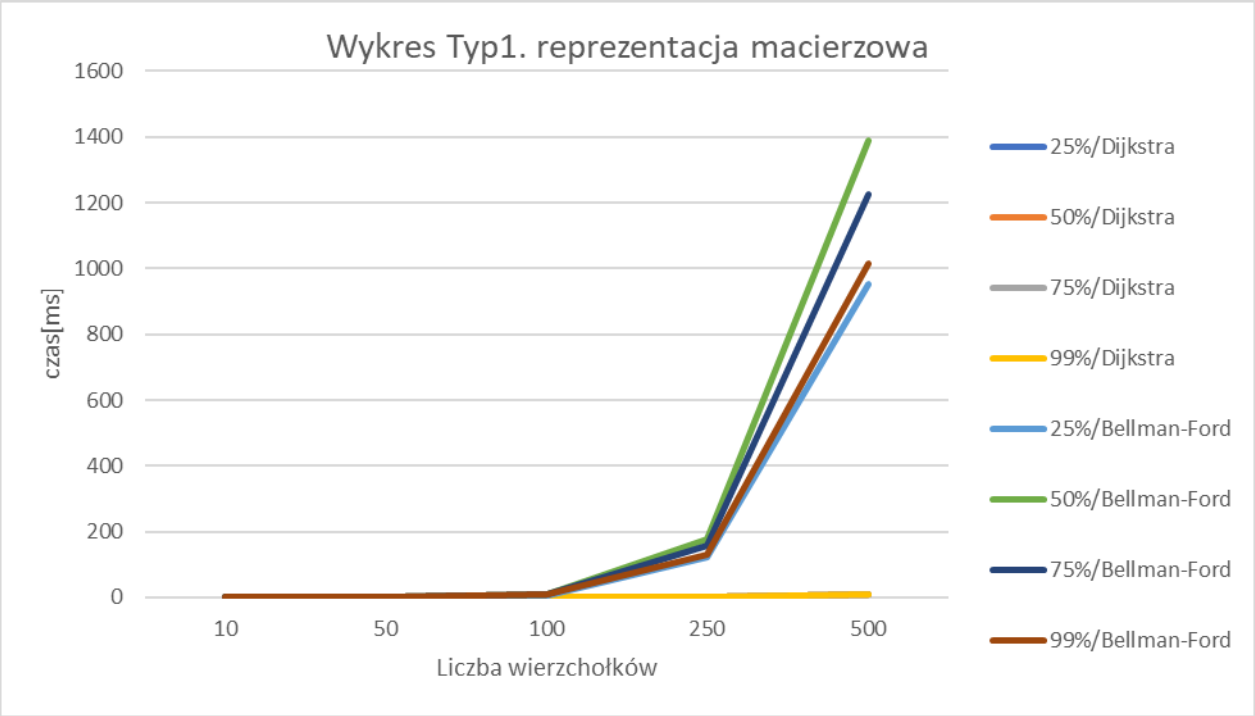
Pierwsza pętla for wykonuje się **V-1** razy, ponieważ iteruje po wszystkich wierzchołkach oprócz ostatniego co zajmuje **O(V).**

W każdej iteracji znajduje wierzchołek o najmniejszej odległości, co zajmuje **O(V^2)** czasu.

Ostateczna złożoność wynosi **O(V^3)**.

gęstość\v	10	50	100	250	500
25%	0,0073	0,8437	7,449	121,32	952,72
50%	0,01	1,2344	10,753	177,47	1388,7
75%	0,0137	1,1679	9,7246	158,98	1226,8
99%	0,0126	1,0225	8,1605	129,68	1015

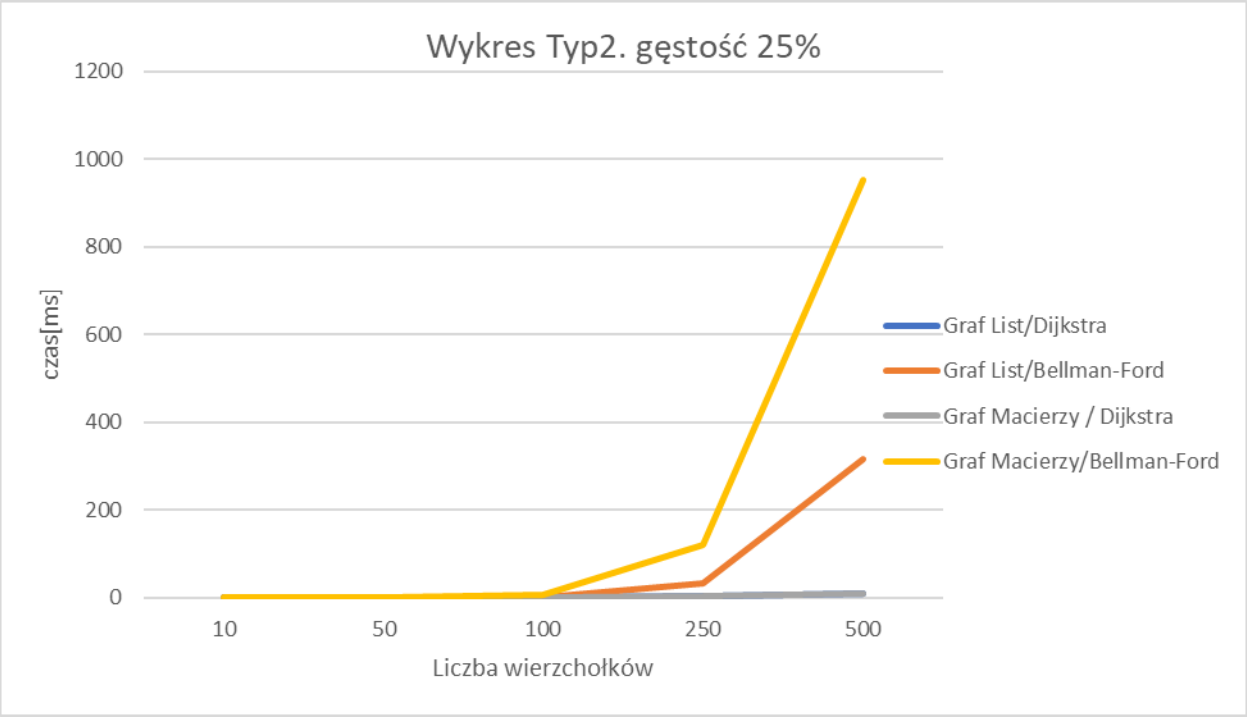
3.2.6. Wykres Typ1. Graf macierzy

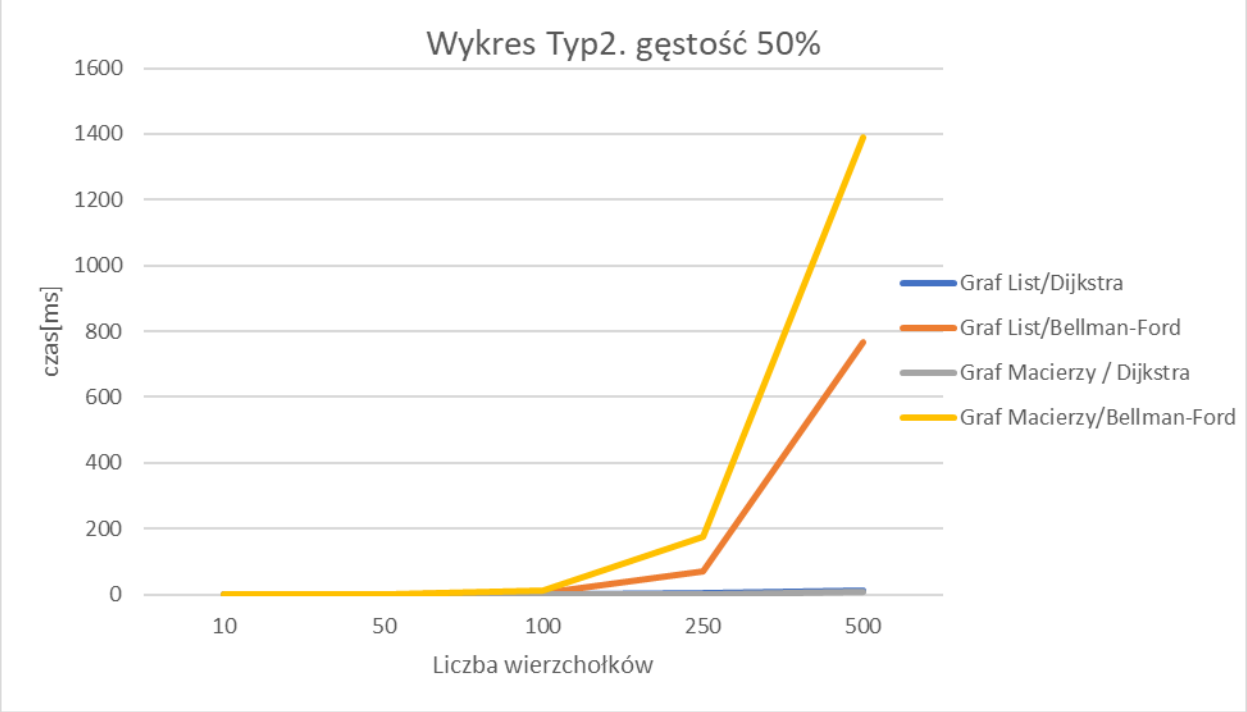


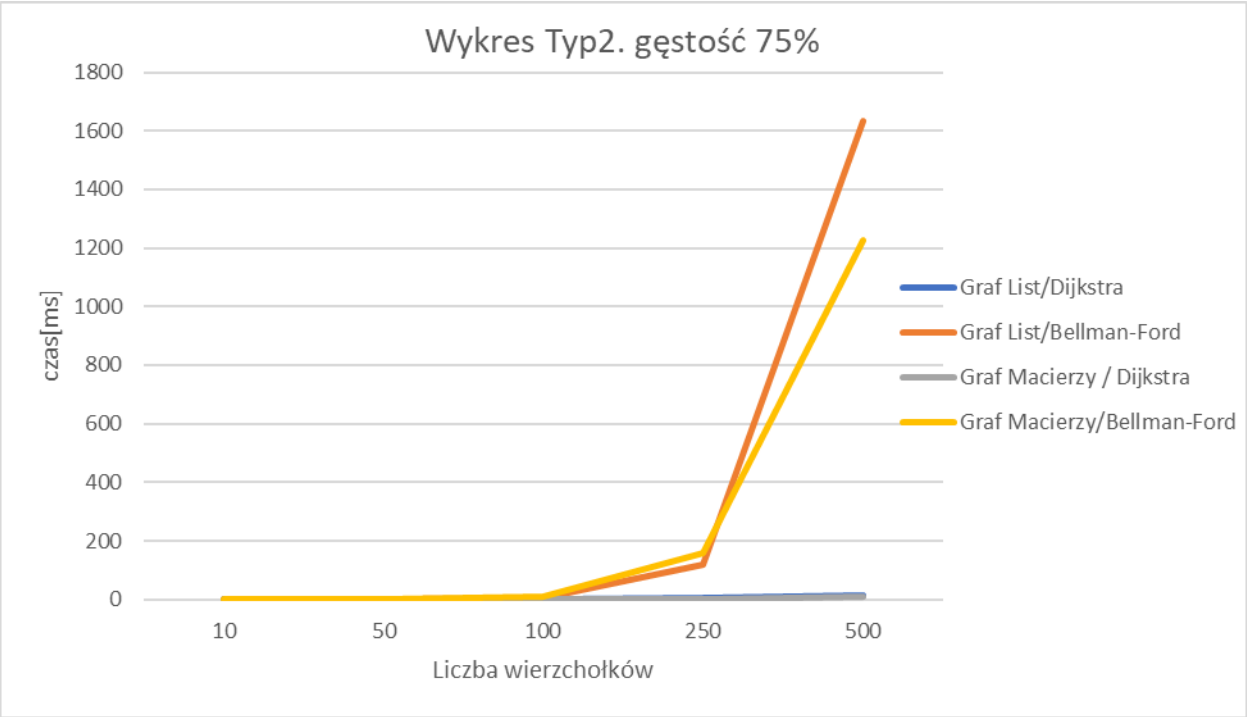
Na wykresie widać znaczącą przewagę algorytmu Dijkstra nad algorytmem Bellmana-Forda. Jednakże algorytm Bellmana-Forda potrafi rozpatrywać cykle ujemne, przez co może być wykorzystany w specyficznych sytuacjach dla grafów z ujemnymi wagami. Ponadto gęstości dla obu algorytmów wpływają na jego wydajność. **O(V^2) < O(V^3).**

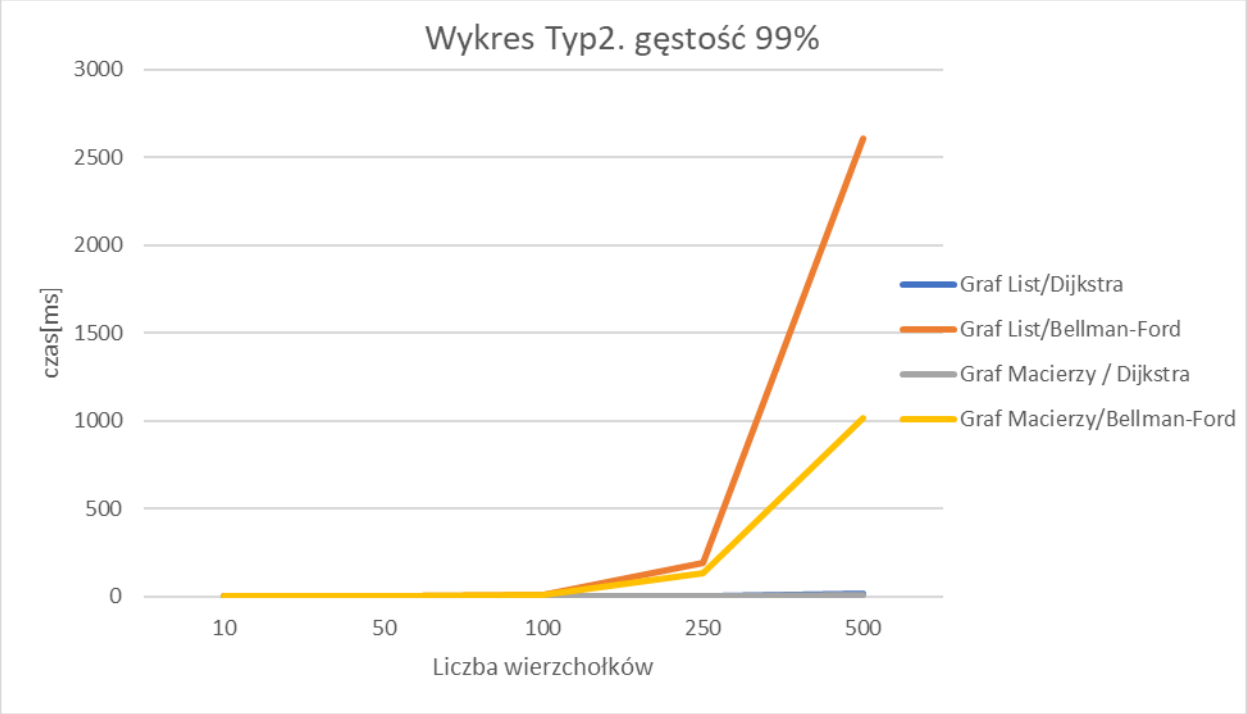
3.2.7. Podsumowanie

Wykresy typ2. parametryzowane typem algorytmu i typem reprezentacji grafu






Wykres Typ2. gęstość 75%
1800
1600
1400
1200
1000
800
600
400
200
0
czas[ms]
10
50
100
250
500
Liczba wierzchołków
Graf List/Dijkstra
Graf List/Bellman-Ford
Graf Macierzy / Dijkstra
Graf Macierzy/Bellman-Ford


Wykres Typ2. gęstość 99%
3000
2500
2000
1500
1000
500
0
czas[ms]
10
50
100
250
500
Liczba wierzchołków
Graf List/Dijkstra
Graf List/Bellman-Ford
Graf Macierzy / Dijkstra
Graf Macierzy/Bellman-Ford

Na wykresach można zauważyć przewagę grafu postaci macierzowej dla wyoskich gęstości, gdzie w praktyce **E** jest bardzo bliskie **V**, gdzie **E** to liczba krawędzi, a **V** to liczba wierzchołków. Oraz przewagę grafu postaci listy sąsiadów dla niższych gęstości. Dzieje się tak, ponieważ iteracja po macierzy jest zawsze kwadratowa, przez co gęstość nie wpływa aż tak znacząco na operacje wykonywane na krawędziach, w przeciwności do listy sąsiadów, gdzie pętla iterująca po kolejnych elementach będzie miała złożoność równą ilości tych elementów.

4. Podsumowanie

Podsumowując, w szczególności możemy zauważyc przewagę grafu w reprezentacji listy sąsiadów, nad grafem reprezentacji macierzowej, dla niższych gęstości. Dzieje się tak ponieważ iteracja po liście jest znacząco szybsza niż iteracja po całej macierzy kwadratowej. Można zauważyć to zjawisko na wykresach typu 2. Gdzie zostały wykazane osobne wykresy dla każdego typu gęstości grafu.