Politechnika Wrocławska

Wydział Informatyki i Telekomunikacji

Kierunek: Informatyczne Systemy Automatyki

Projektowanie i analiza algorytmów

Michał Wróblewski

272488

Termin zajęć:

wtorek 13:15 - 15:00

17 kwietnia 2024

Spis treści

Spis tabel

Spis rysunków

1 Wstęp

W tym sprawozdaniu przedstawione są wyniki badań dotyczących efektywności algorytmu Dijkstry w zależności od sposobu reprezentacji, gęstości oraz ilości wierzchołków grafu.

2 Cel

Celem badań było zbadanie efektywności algorytmu Dijkstry dla różnych sposobów reprezentacji grafu (macierz i lista) oraz różnych gęstości grafu.

3 Metodologia

Badania zostały przeprowadzone dla 5 różnych liczb wierzchołków oraz dla następujących gęstości grafu: 25%, 50%, 75% oraz dla grafu pełnego. Dla każdego zestawu reprezentacji grafu, liczby wierzchołków i gęstości wygenerowano po 100 losowych instancji. Następnie uśredniono wyniki.

4 Algorytm Dijkstry

Algorytm Dijkstry służy do znajdowania najkrótszych ścieżek w grafach z wagami na krawędziach. Oto jak działa ten algorytm:

1. **Inicjalizacja**: Na początku ustawiamy odległość do wszystkich wierzchołków na nieskończoność, z wyjątkiem wierzchołka startowego, którego odległość ustawiamy na 0. Dodajemy startowy wierzchołek do kolejki priorytetowej.

2. **Iteracja**: W każdej iteracji wybieramy z kolejki wierzchołek o najmniejszej odległości. Następnie sprawdzamy wszystkie jego sąsiadujące krawędzie. Jeśli sumaryczna odległość od źródła do sąsiada przez aktualny wierzchołek jest mniejsza od obecnej odległości do sąsiada, aktualizujemy odległość do tego sąsiada.

3. **Zakończenie**: Gdy wszystkie krawędzie zostaną sprawdzone i zaktualizowane, otrzymujemy najkrótsze ścieżki do wszystkich wierzchołków od źródła.

4.1 Kod Algorytmu Dijkstry

4.1.1 Macierz sąsiedztwa

Kod algorytmu Dijkstry dla reprezentacji grafu za pomocą macierzy sąsiedztwa:

```
double dijkstra(int source, ChangeablePriorityQueue& pq) {
    auto start = chrono::high_resolution_clock::now();  // O(1)
    if (source < 0 || source >= V) {  // O(1)
        cout << "Invalid source vertex index!" << endl;  // O(1)
        return 0.0;  // O(1)
    }

    int* distance = new int[V];  // O(V)
```

```
    bool* visited = new bool[V];  // O(V)

    for (int i = 0; i < V; ++i) {  // O(V)
        distance[i] = numeric_limits<int>::max();  // O(1)
        visited[i] = false;  // O(1)
    }
    distance[source] = 0;  // O(1)

    pq.build(new Target(source, 0));  // O(log V)

    while (!pq.isEmpty()) {  // O(V^2)
        Target* current = pq.peek();  // O(1)
        int u = current->vertex;  // O(1)
        pq.deleteMin();  // O(log V)

        if (visited[u]) continue;  // O(1)

        visited[u] = true;  // O(1)

        for (int v = 0; v < V; ++v) {  // O(V)
            if (!visited[v] && adjacencyMatrix[u][v] != -1 && distance[u] !=
             ↪  numeric_limits<int>::max() &&
                distance[u] + adjacencyMatrix[u][v] < distance[v]) {  // O(1)
                distance[v] = distance[u] + adjacencyMatrix[u][v];  // O(1)

                int index = -1;  // O(1)
                if (pq.find(v)) {  // O(log V)
                    index = v;  // O(1)
                }

                if (index != -1) {  // O(1)
                    pq.decreaseKey(index, distance[v]);  // O(log V)
                }
                else {
                    pq.build(new Target(v, distance[v]));  // O(log V)
                }
            }
        }
        delete current;  // O(1)
    }
    auto end = chrono::high_resolution_clock::now();  // O(1)
    return chrono::duration<double, milli>(end - start).count();  // O(1)

    delete[] distance;  // O(1)
    delete[] visited;  // O(1)
}
```

4.1.2 Lista sąsiedztwa

Kod algorytmu Dijkstry dla reprezentacji grafu za pomocą listy sąsiedztwa:

```
double dijkstra(int source, ChangeablePriorityQueue& pq) {
    auto start = chrono::high_resolution_clock::now();  // O(1)

    if (source < 0 || source >= V) {  // O(1)
        cout << "Invalid source vertex index!" << endl;  // O(1)
        return 0.0;  // O(1)
    }

    int* distance = new int[V];  // O(V)
    bool* visited = new bool[V];  // O(V)

    for (int i = 0; i < V; ++i) {  // O(V)
        distance[i] = numeric_limits<int>::max();  // O(1)
        visited[i] = false;  // O(1)
    }
    distance[source] = 0;  // O(1)

    pq.build(new Target(source, 0));  // O(log V)

    while (!pq.isEmpty()) {  // O((V + E) * log V)
        Target* current = pq.peek();  // O(1)
        int u = current->vertex;  // O(1)
        pq.deleteMin();  // O(log V)

        if (visited[u]) continue;  // O(1)

        visited[u] = true;  // O(1)

        for (int i = 0; i < V; i++) {  // O(V)
            const auto& neighbor = adjacencyList[u][i];  // O(1)
            int v = neighbor.first;  // O(1)
            int weight = neighbor.second;  // O(1)

            if (!visited[v] && distance[u] != numeric_limits<int>::max() &&
                distance[u] + weight < distance[v]) {  // O(1)
                distance[v] = distance[u] + weight;  // O(1)

                int index = -1;  // O(1)
                if (pq.find(v)) {  // O(log V)
                    index = v;  // O(1)
                }

                if (index != -1) {  // O(1)
                    pq.decreaseKey(index, distance[v]);  // O(log V)
                }
```

```
                else {
                    pq.build(new Target(v, distance[v]));  // O(log V)
                }
            }
        }
        delete current;  // O(1)
    }
    auto end = chrono::high_resolution_clock::now();  // O(1)
    return chrono::duration<double, milli>(end - start).count();  // O(1)

    delete[] distance;  // O(1)
    delete[] visited;  // O(1)
}
```

5 Złożoność obliczeniowa

5.1 Wykorzystanie kopca binarnego na tablicy dynamicznej

Wykorzystanie kopca binarnego na tablicy dynamicznej do implementacji kolejki priorytetowej stanowi efektywne rozwiązanie w kontekście algorytmu Dijkstry. Dzięki temu podejściu możliwe jest szybkie dodawanie i usuwanie elementów z kolejki, co jest kluczowe podczas *relaksacji krawędzi* w trakcie wykonywania algorytmu.

Kopiec binarny na tablicy dynamicznej umożliwia elastyczne dostosowanie rozmiaru kolejki w miarę dodawania i usuwania elementów. W każdym momencie operacje wstawiania (`insert`) i usuwania (`extractMin`) mają złożoność czasową $O(\log n)$, gdzie n to liczba elementów w kolejce.

Dzięki temu rozwiązaniu można efektywnie przechowywać wierzchołki w kolejce priorytetowej, co przyspiesza działanie algorytmu Dijkstry poprzez szybkie wybieranie wierzchołka o najmniejszej odległości oraz szybkie aktualizowanie odległości do wierzchołków w kolejce.

Implementacja kopca binarnego na tablicy dynamicznej umożliwia optymalne wykorzystanie zasobów pamięciowych oraz zapewnia szybki dostęp do elementów kolejki priorytetowej, co sprawia, że jest to moim zdaniem idealne rozwiązanie w kontekście algorytmu Dijkstry.

5.2 Złożoność obliczeniowa

W przypadku implementacji algorytmu Dijkstry z wykorzystaniem kolejki priorytetowej opartej na kopcu binarnym na tablicy dynamicznej, złożoność obliczeniowa wynosi $O((V + E)\log V)$, gdzie V to liczba wierzchołków, a E to liczba krawędzi w grafie.

Operacje dodawania wierzchołków do kolejki oraz usuwania wierzchołków z kolejki mają złożoność $O(\log V)$, a w najgorszym przypadku mogą zostać wykonane dla każdej z E krawędzi. Dlatego złożoność obliczeniowa algorytmu wynosi $O((V + E)\log V)$.

Należy również zauważyć, że w przypadku zastosowania reprezentacji grafu za pomocą *macierzy sąsiedztwa*, złożoność obliczeniowa operacji relaksacji krawędzi wynosi $O(V^2)$. Jednakże przy wykorzystaniu *listy sąsiedztwa* ta złożoność jest zależna od liczby krawędzi w grafie i wynosi $O(E)$.

Podsumowując, zastosowanie kopca binarnego na tablicy dynamicznej do implementacji kolejki priorytetowej w algorytmie Dijkstry pozwala na efektywne znajdowanie najkrótszych ścieżek w grafie, przy zachowaniu złożoności obliczeniowej $O((V + E)\log V)$.

6 Losowanie krawędzi

Funkcja *addRandomEdges* generuje krawędzie grafu na podstawie zadanej gęstości oraz maksymalnej wagi.

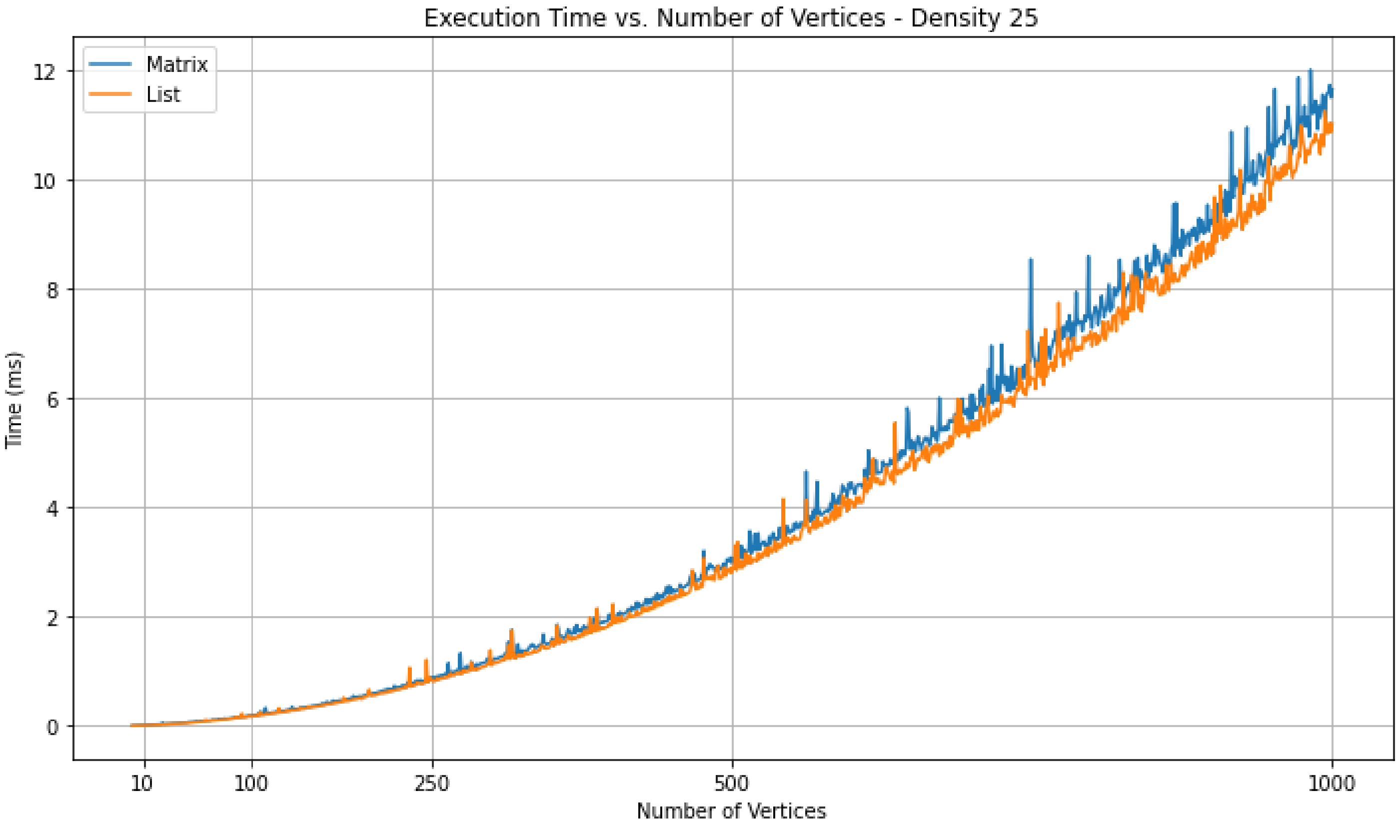
Dodawane krawędzie są dwukierunkowe, co zapewnia symetryczność grafu.

```
void addRandomEdges(int density, int maxWeight) {
    srand(time(0));
    for (int i = 0; i < V; ++i) {
        for (int j = 0; j < V; ++j) {
            if (i != j) { // bo d(s,s) jest rowne 0
                if (rand() % 100 < density) {
                    int weight = rand() % maxWeight + 1;
                    adjacencyList.addEdge(i, j, weight);
                    adjacencyList.addEdge(j, i, weight);
                    adjacencyMatrix.addEdge(i, j, weight);
                    adjacencyMatrix.addEdge(j, i, weight);
                }
            }
        }
    }
}
```

7 Wyniki

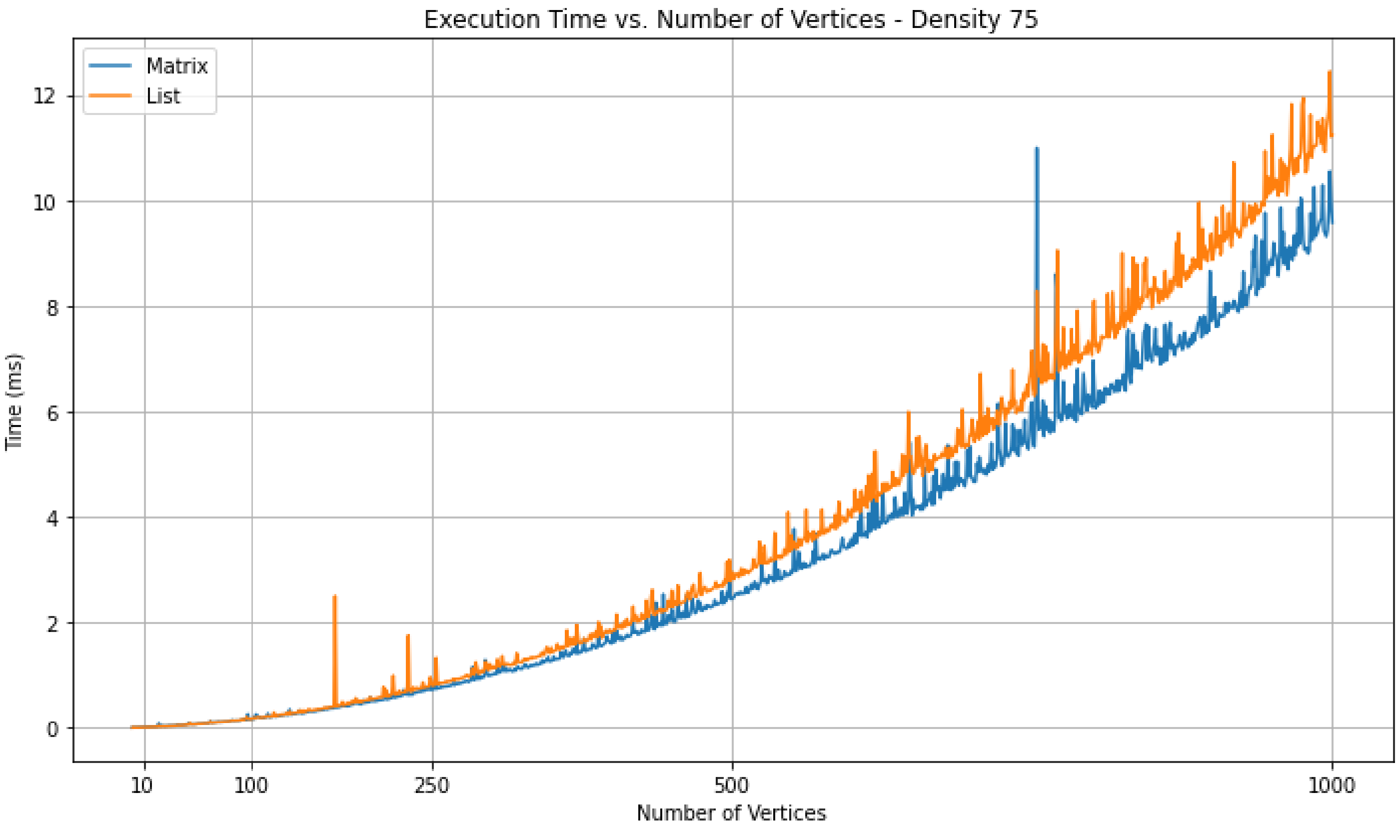
7.1 Większy zakres wartości wierzchołków [10, 100, 250, 500, 1000]

7.1.1 Czas w zależności od ilości wierzchołków grafu - Wykresy

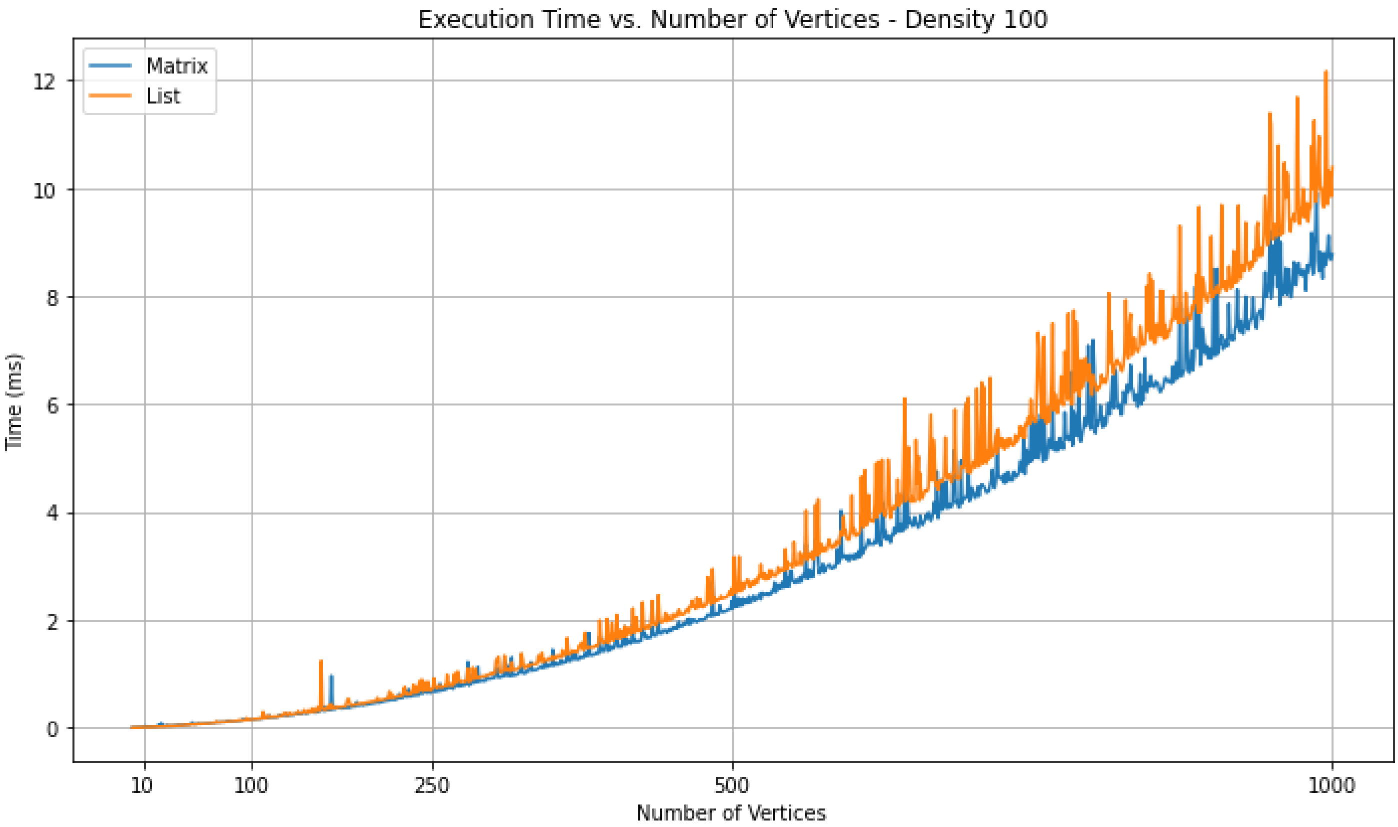


Rysunek 1: Czas wykonania w zależności od liczby wierzchołków dla gęstości grafu równej 25

Rysunek 2: Czas wykonania w zależności od liczby wierzchołków dla gęstości grafu równej 50

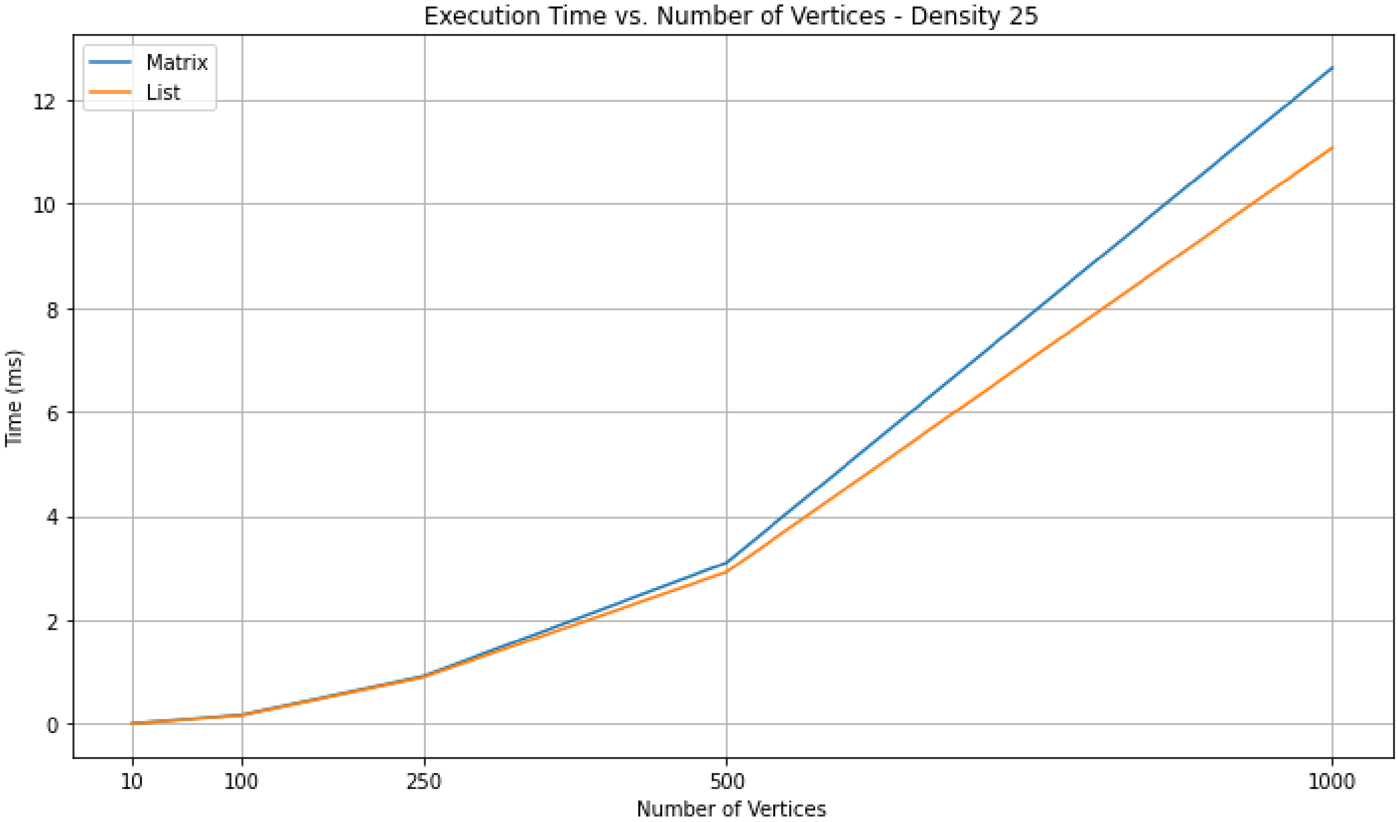


Rysunek 3: Czas wykonania w zależności od liczby wierzchołków dla gęstości grafu równej 75

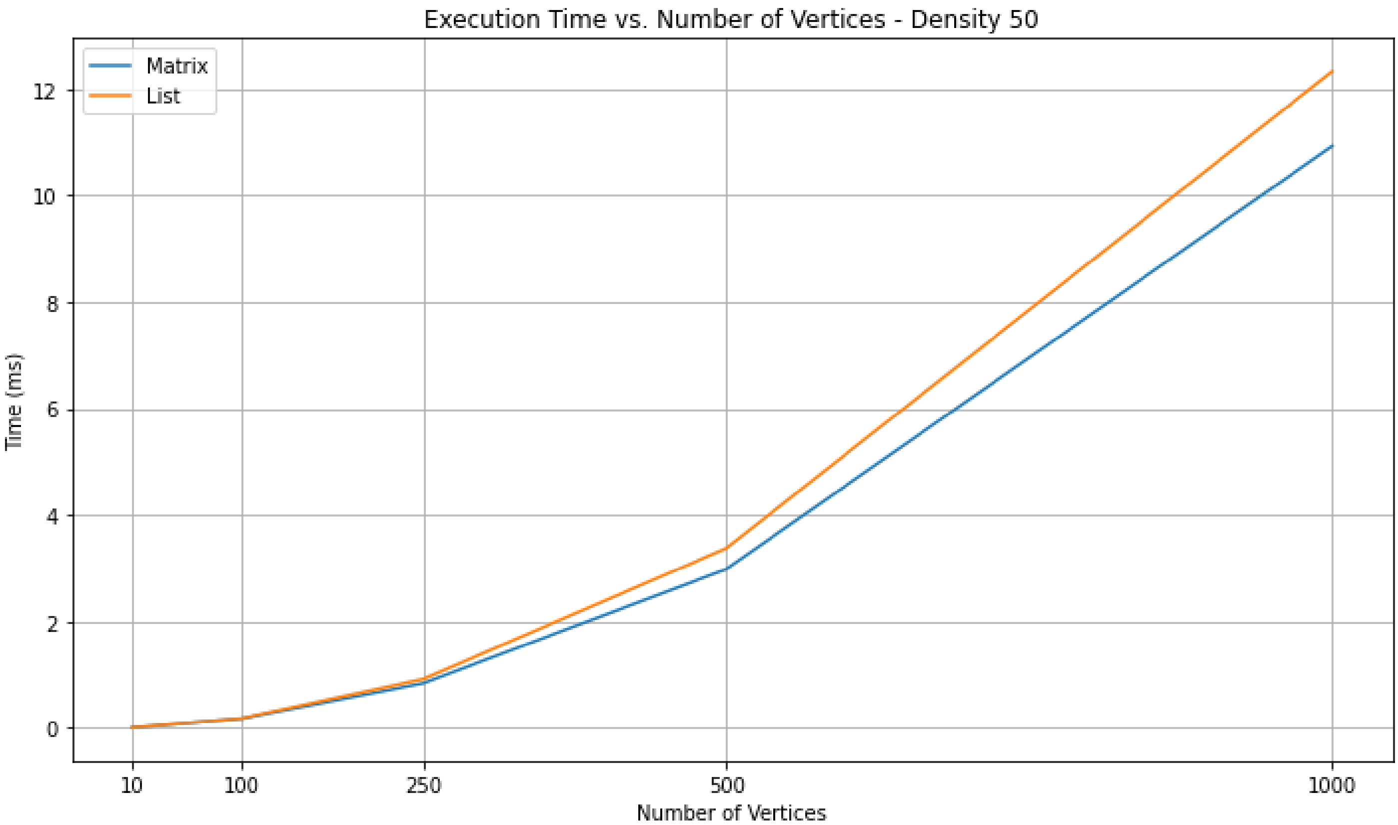


Rysunek 4: Czas wykonania w zależności od liczby wierzchołków dla gęstości grafu równej 100

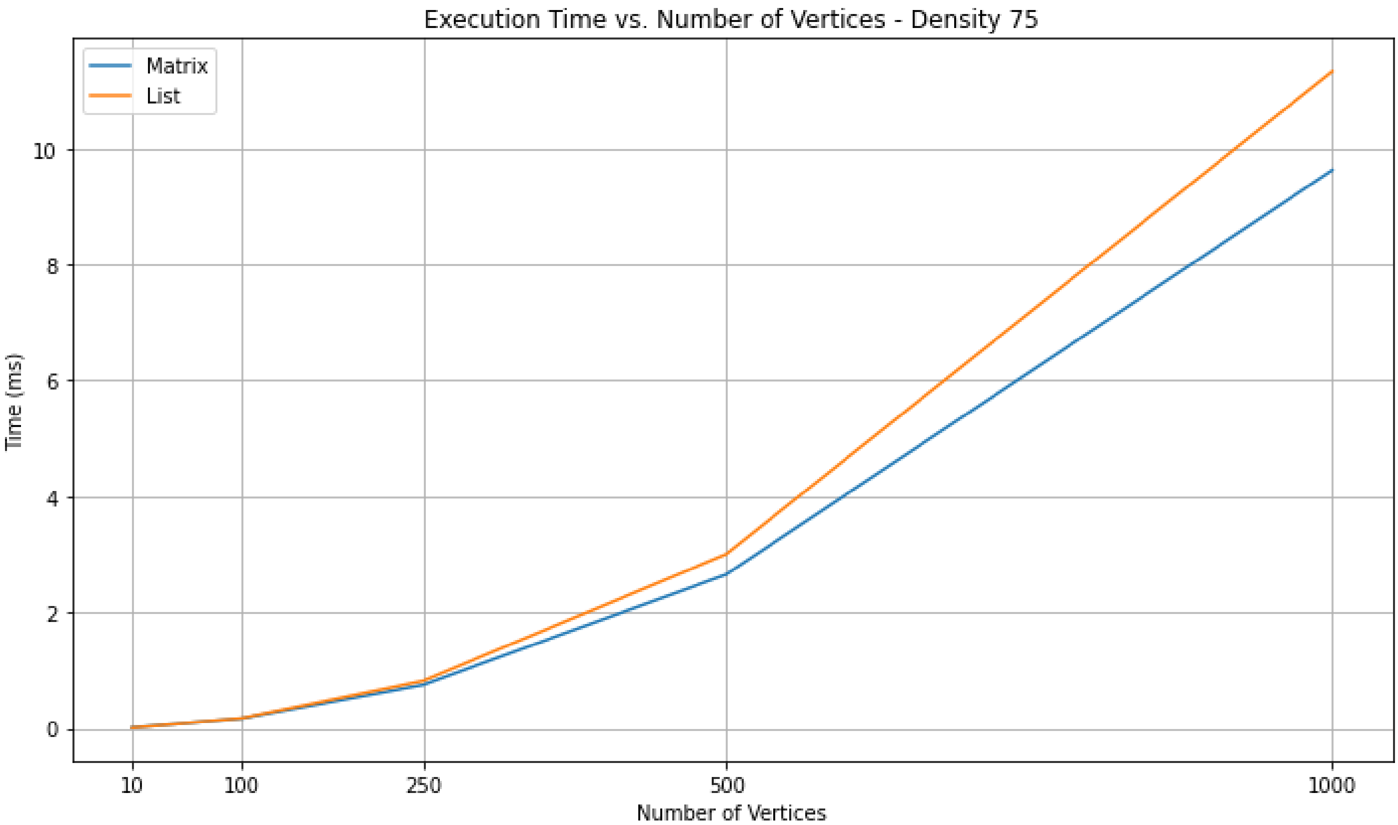
7.1.2 Czas w zależności od ilości wierzchołków grafu - Wykresy uśrednione



Rysunek 5: Czas wykonania w zależności od liczby wierzchołków dla gęstości grafu równej 25



Rysunek 6: Czas wykonania w zależności od liczby wierzchołków dla gęstości grafu równej 50



Rysunek 7: Czas wykonania w zależności od liczby wierzchołków dla gęstości grafu równej 75

Rysunek 8: Czas wykonania w zależności od liczby wierzchołków dla gęstości grafu równej 100

7.1.3 Czas w zależności od ilości wierzchołków grafu - Tabele

Number of Vertices	Matrix Time (ms)	List Time (ms)
10	0,0079	0,0048
100	0,1805	0,1715
250	0,9192	0,9006
500	3,0966	2,9160
1000	12,6087	11,0728

Tabela 1: Czas wykonania dla gęstości 25

Number of Vertices	Matrix Time (ms)	List Time (ms)
10	0,0125	0,0075
100	0,1793	0,1846
250	0,8430	0,9231
500	2,9847	3,3758
1000	10,9296	12,3310

Tabela 2: Czas wykonania dla gęstości 50

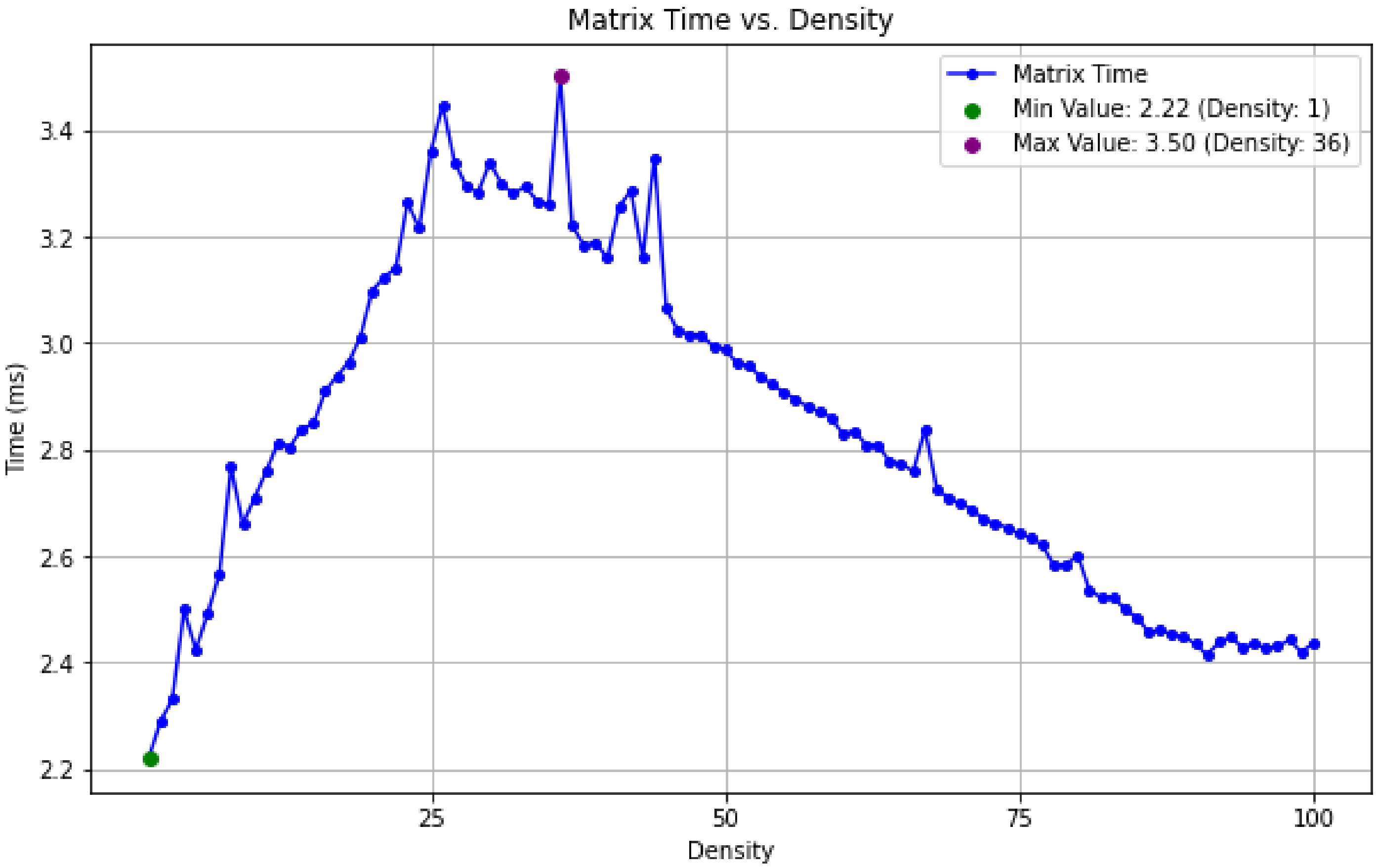
Number of Vertices	Matrix Time (ms)	List Time (ms)
10	0,0121	0,0085
100	0,1677	0,1725
250	0,7480	0,8171
500	2,6525	3,0016
1000	9,6285	11,3332

Tabela 3: Czas wykonania dla gęstości 75

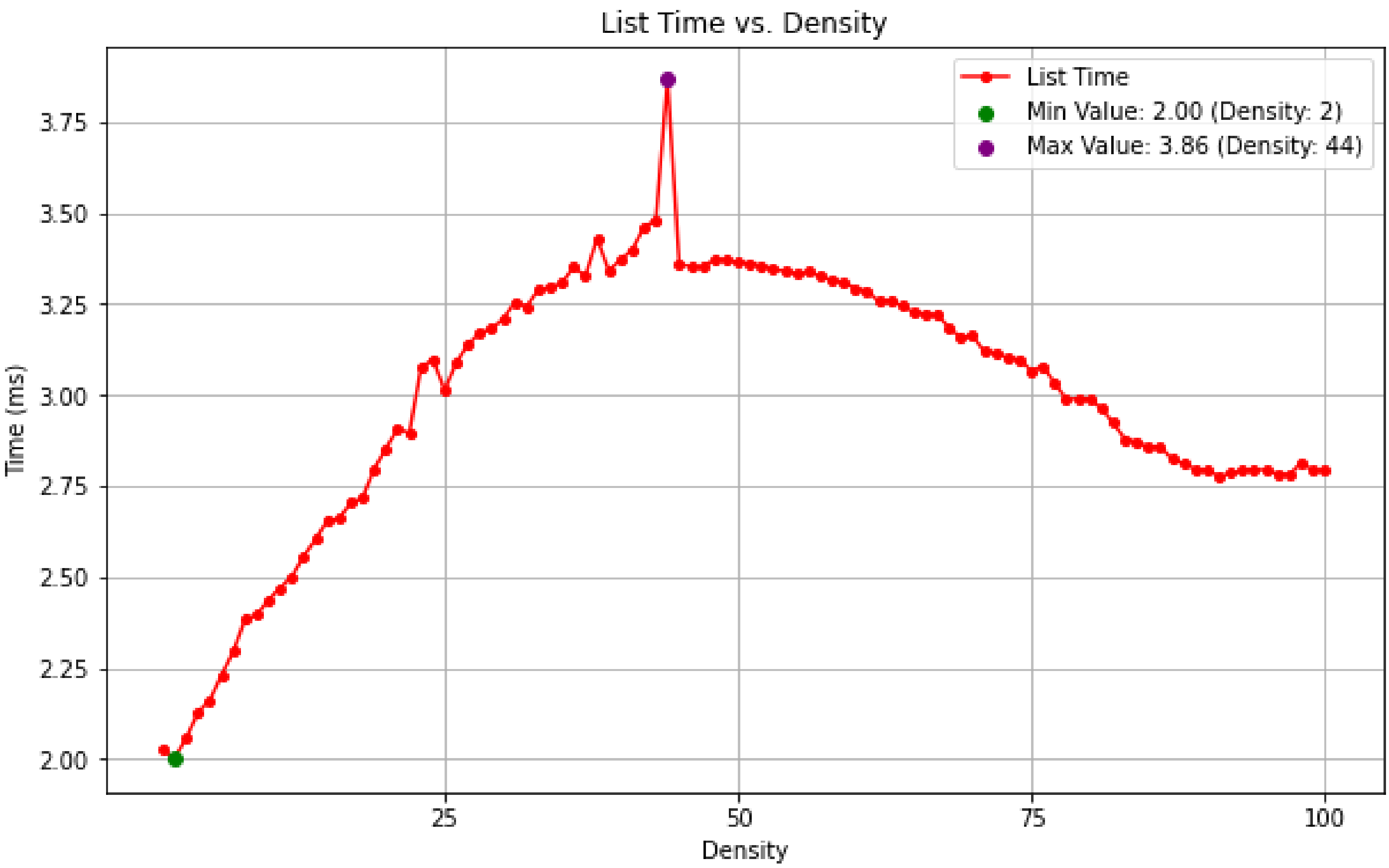
Number of Vertices	Matrix Time (ms)	List Time (ms)
10	0,0093	0,0063
100	0,1602	0,1577
250	0,6777	0,7387
500	2,4804	2,8028
1000	8,8480	10,2635

Tabela 4: Czas wykonania dla gęstości 100

7.1.4 Czas w zależności od gęstości grafu - Wykresy



Rysunek 9: Czas wykonania na macierzy sąsiadującej w zależności od gęstości



Rysunek 10: Czas wykonania na liście sąsiadującej w zależności od gęstości

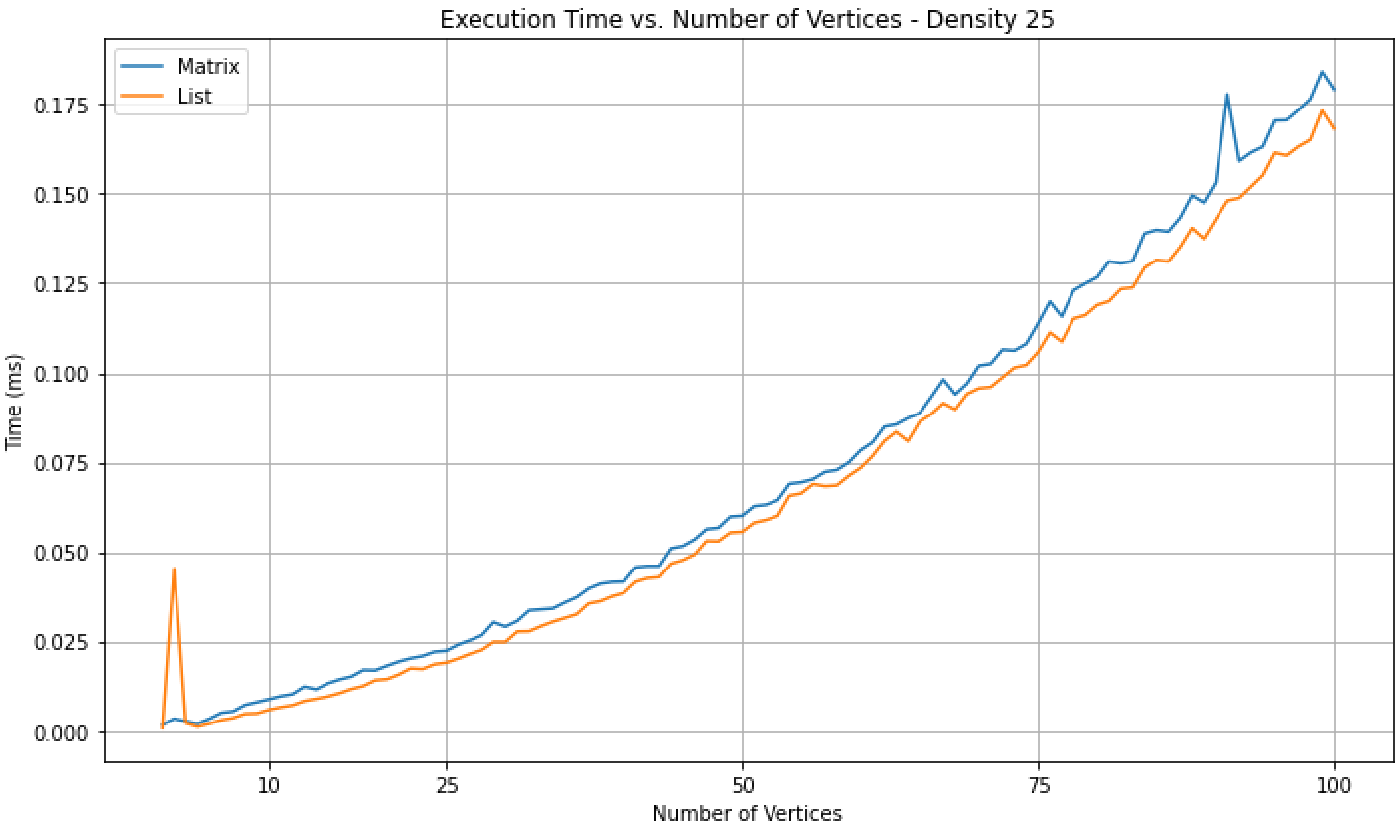
7.1.5 Czas w zależności od gęstości grafu - Tabele

Density	Matrix Time (ms)	List Time (ms)
25	3,3626	3,0132
50	2,9898	3,3644
75	2,6418	3,0666
100	2,4351	2,7938

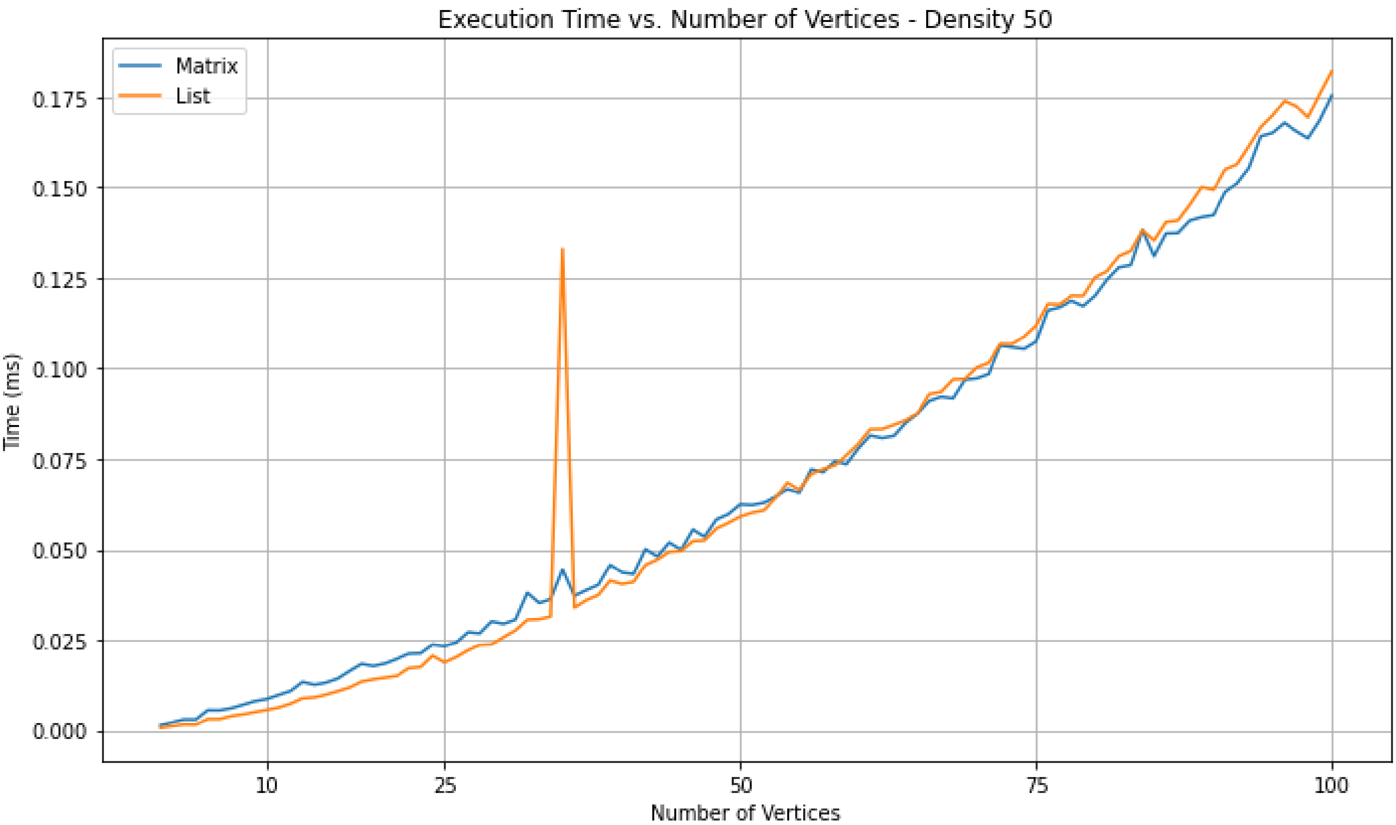
Tabela 5: Średni czas wykonania w zależności od gęstości grafu

7.2 Mniejszy zakres wartości wierzchołków [10, 25, 50, 75, 100]

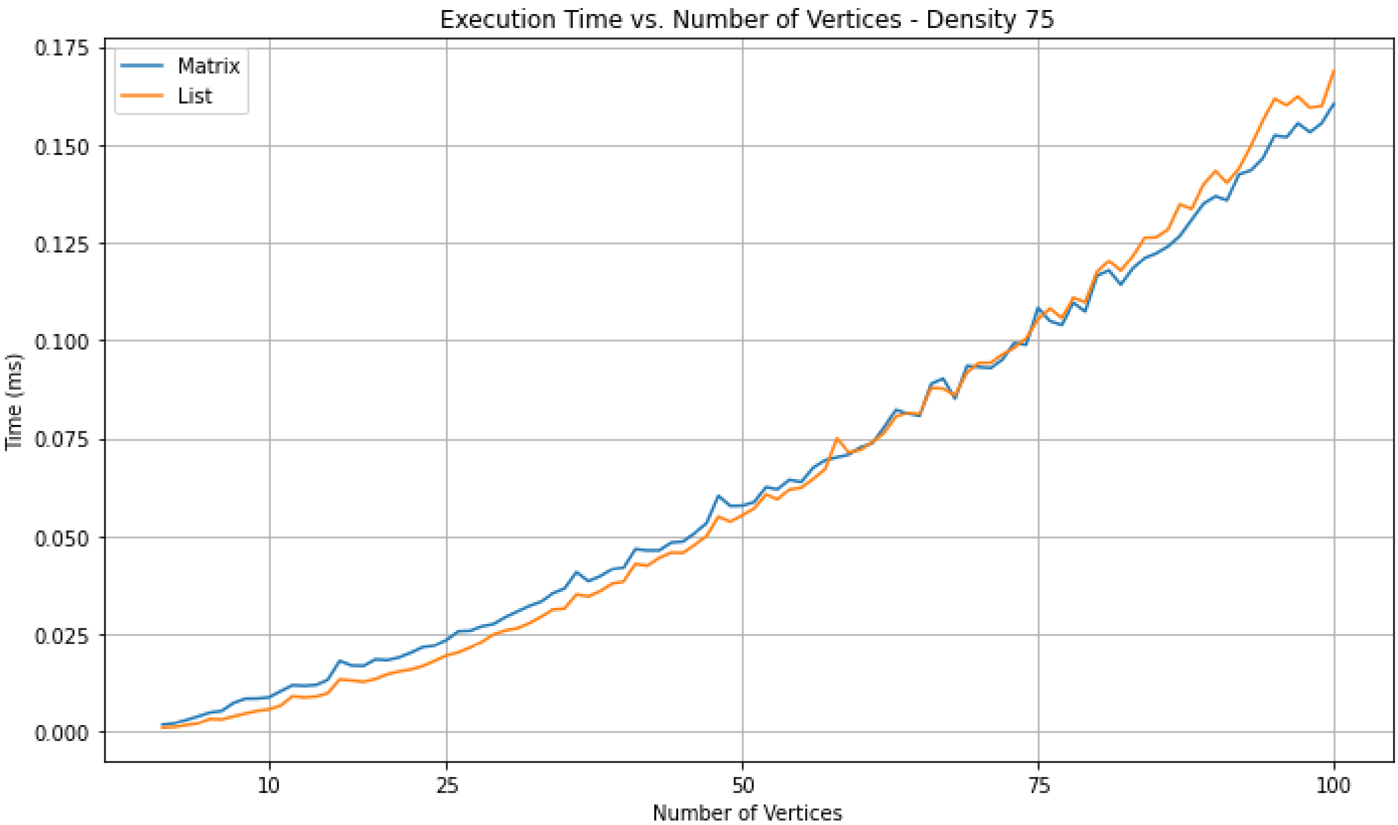
7.2.1 Czas w zależności od ilości wierzchołków grafu - Wykresy



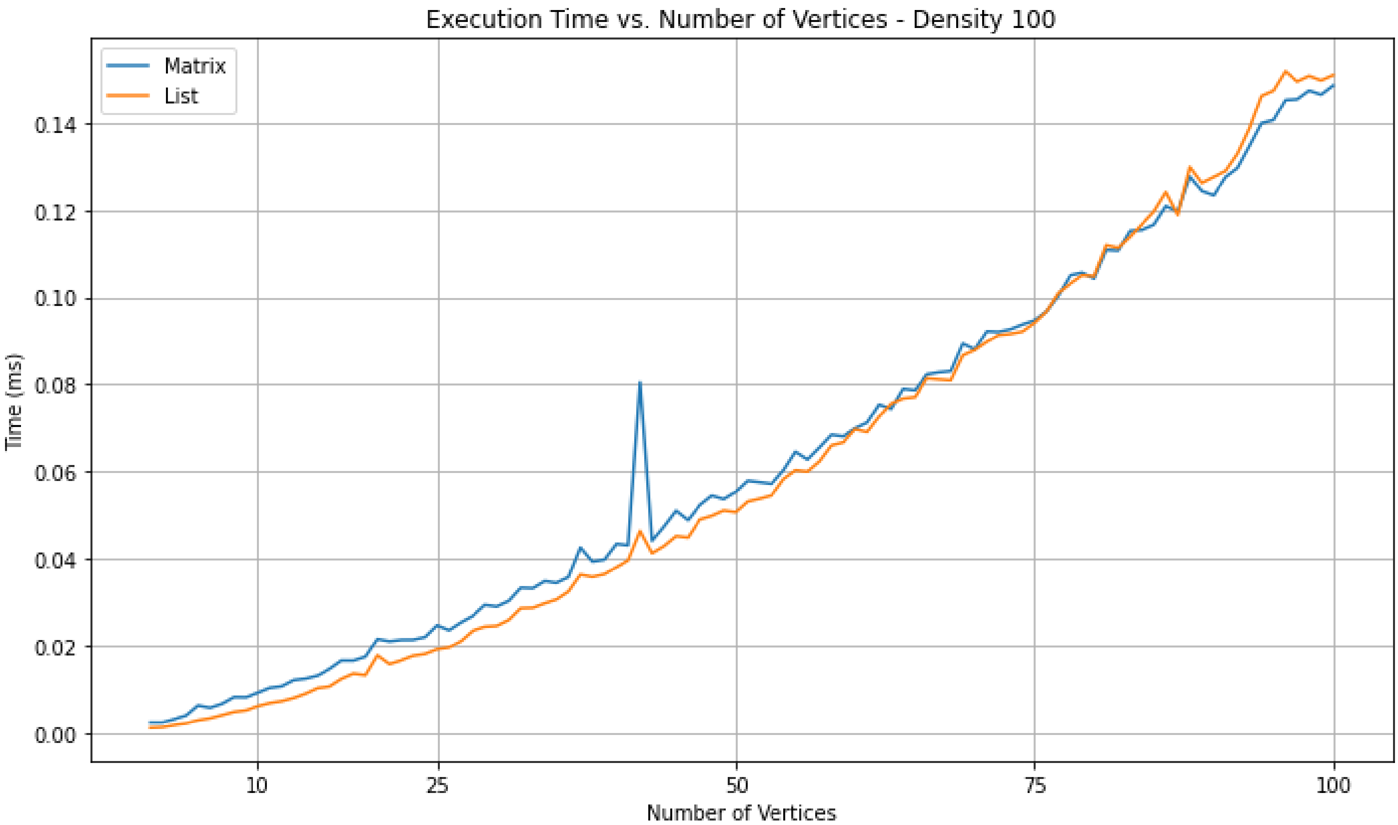
Rysunek 11: Czas wykonania w zależności od liczby wierzchołków dla gęstości grafu równej 25



Rysunek 12: Czas wykonania w zależności od liczby wierzchołków dla gęstości grafu równej 50



Rysunek 13: Czas wykonania w zależności od liczby wierzchołków dla gęstości grafu równej 75



Rysunek 14: Czas wykonania w zależności od liczby wierzchołków dla gęstości grafu równej 100

7.2.2 Czas w zależności od ilości wierzchołków grafu - Tabele

Vertices	Matrix Time (ms)	List Time (ms)
10	0,0089	0,0059
25	0,0224	0,0191
50	0,0601	0,0556
75	0,1138	0,1058
100	0,1792	0,1682

Tabela 6: Execution Time for Density 25

Vertices	Matrix Time (ms)	List Time (ms)
10	0,0087	0,0057
25	0,0233	0,0188
50	0,0624	0,0591
75	0,1074	0,1118
100	0,1754	0,1820

Tabela 7: Execution Time for Density 50

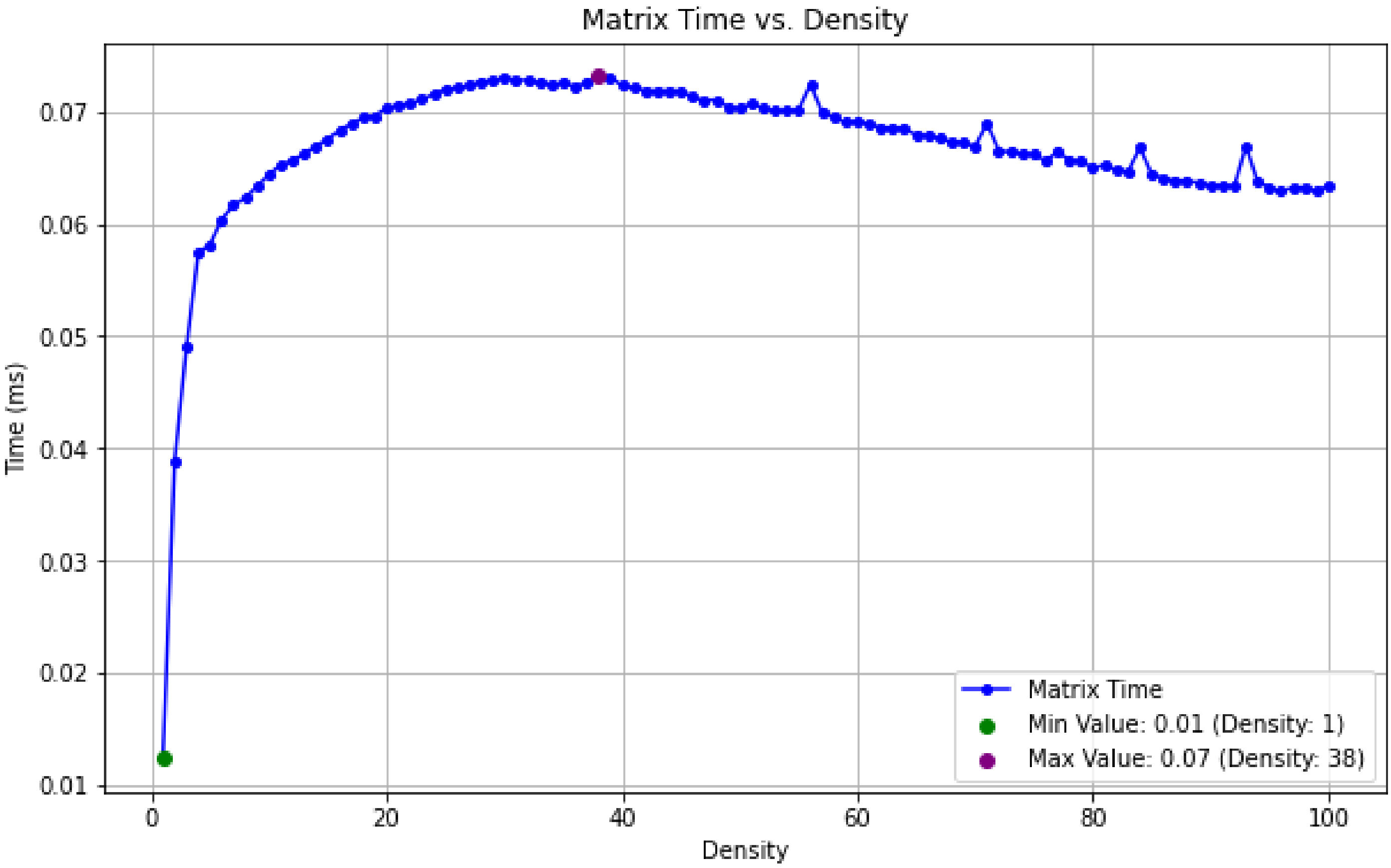
Vertices	Matrix Time (ms)	List Time (ms)
10	0,0088	0,0057
25	0,0235	0,0194
50	0,0577	0,0552
75	0,1084	0,1054
100	0,1605	0,1688

Tabela 8: Execution Time for Density 75

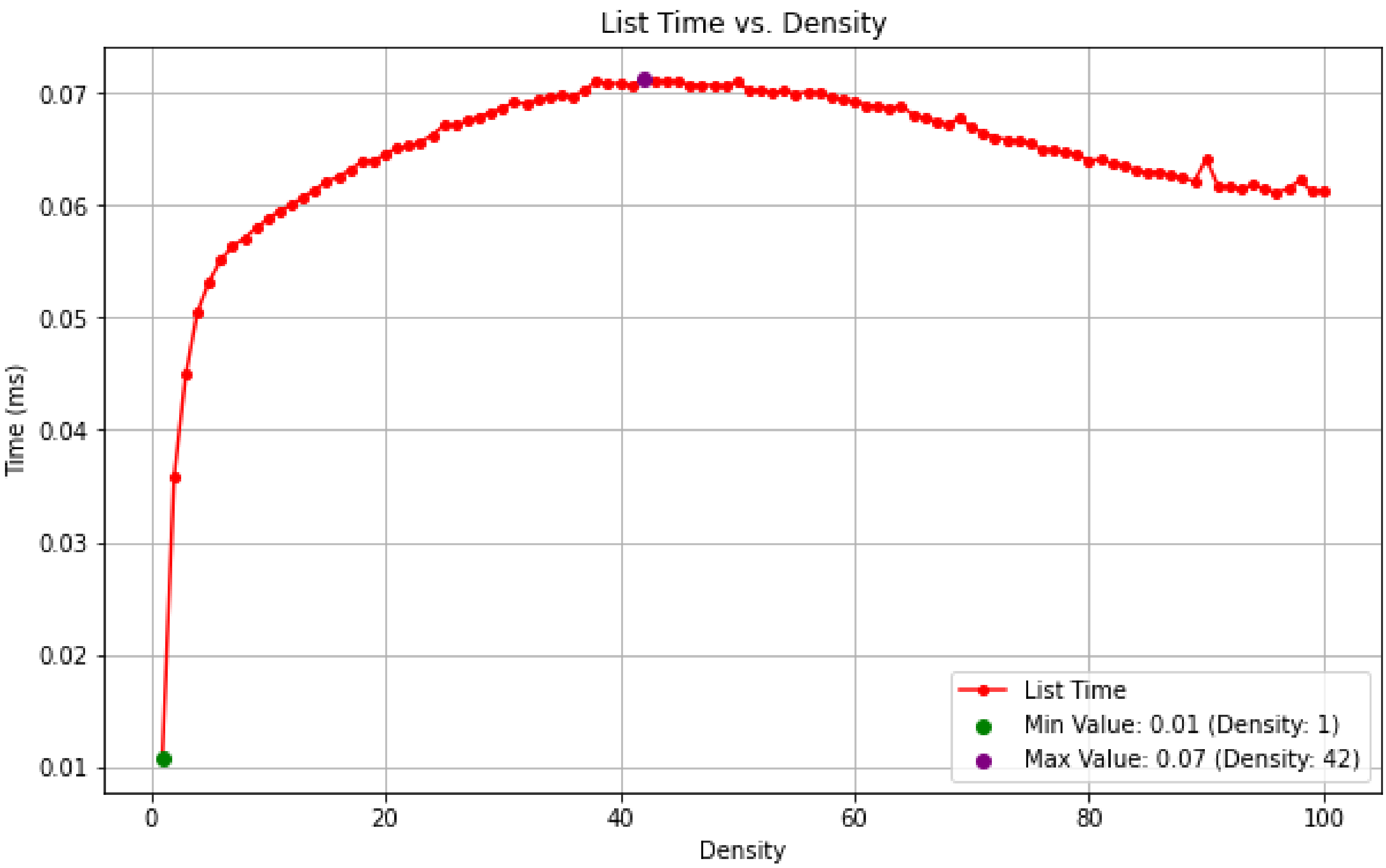
Vertices	Matrix Time (ms)	List Time (ms)
10	0,0092	0,0061
25	0,0246	0,0191
50	0,0553	0,0506
75	0,0947	0,0941
100	0,1486	0,1510

Tabela 9: Execution Time for Density 100

7.2.3 Czas w zależności od gęstości grafu - Wykresy



Rysunek 15: Czas wykonania na macierzy sąsiadującej w zależności od gęstości



Rysunek 16: Czas wykonania na liście sąsiadującej w zależności od gęstości

7.2.4 Czas w zależności od gęstości grafu - Tabele

Density	Matrix Time (ms)	List Time (ms)
25	0,0720	0,0671
50	0,0703	0,0710
75	0,0662	0,0655
100	0,0635	0,0613

Tabela 10: Średni czas wykonania w zależności od gęstości grafu

8 Podsumowanie

W niniejszym sprawozdaniu przeprowadzono badania nad efektywnością algorytmu Dijkstry w zależności od różnych sposobów reprezentacji grafu i gęstości grafu. Wyniki badań potwierdziły, że gęstość grafu ma znaczący wpływ na czas wykonania algorytmu: im większa gęstość, tym krótszy czas działania.

Analiza czasu wykonania algorytmu w zależności od gęstości grafu wykazała interesujące zależności.

Dla gęstości grafu wynoszącej 25, czas wykonania algorytmu na macierzy sąsiedztwa był dłuższy niż czas wykonania algorytmu na liście sąsiedztwa. Jednakże, ten trend odwrócił się dla gęstości 50, 75 i 100, gdzie implementacja jako macierz sąsiedztwa osiągnęła lepsze wyniki czasowe od listy sąsiedztwa.

Warto zauważyć, że czas wykonania dla listy sąsiedztwa przy gęstości 25 był szybszy niż przy gęstościach 50 i 75.

Natomiast, analizując czas wykonania w zależności od liczby wierzchołków grafu, obserwujemy, że lista sąsiedztwa jako implementacja grafu okazała się szybsza od macierzy sąsiedztwa.

Ponadto, im większa gęstość grafu, tym bardziej zbliżone są do siebie wyniki czasowe dla obu implementacji. Te wnioski sugerują istotność wyboru odpowiedniej reprezentacji grafu w zależności od jego gęstości oraz liczby wierzchołków, aby zoptymalizować czas wykonania algorytmów.

Podsumowując, wybór między macierzą a listą sąsiedztwa zależy od charakterystyki konkretnego grafu oraz wymagań dotyczących zużycia pamięci i czasu wykonania algorytmów. Dla grafów pełnych macierz sąsiedztwa może być bardziej korzystna, podczas gdy dla grafów o dużej liczbie wierzchołków lista sąsiedztwa może zapewnić lepszą wydajność.