Projektowanie algorytmów i metody sztucznej inteligencji

Projekt 2

Termin zajęć: Wykonał:

Czwartek 9:15 Eryk Matecki 249484

Prowadzący: dr inż. Łukasz Jeleń

1. Wstęp

Celem projektu było przetestowanie działania algorytmów wyznaczających najkrótszą drogę z danego węzła grafu do pozostałych węzłów. Algorytmem użytym i testowanym przeze mnie był algorytm Bellmana-Forda. Testowane przeze mnie grafy różniły się gęstością, czyli ilością krawędzi, ilością węzłów oraz reprezentacją grafów. Grafy były reprezentowane na dwa sposoby:

- Lista sąsiedztwa,
- Macierz sąsiedztwa.

Badania były wykonywane dla 5 różnych liczb wierzchołków w grafie V:

- 10,
- 50,
- 100,
- 500,
- 1000,

oraz następujących gęstości grafu:

- 25%,
- 50%
- 75%,
- graf pełny.

Dla każdego takiego zestawu wygenerowano 100 prób by móc bardziej sprecyzować wyniki, które następnie zostały uśrednione w celu pokazania wyników. Program również umożliwia wczytywanie grafów z plików tekstowych oraz możliwość zapisu wyniku działania algorytmu. Zapis danych w pliku powinien wyglądać następująco:

ilość_krawędzi ilość_wierzchołków wierzchołek_startowy

wierzchołek_początkowy wierzchołek_końcowy waga

.

.

.

Po wykonaniu się algorytmu i znalezieniu najkrótszej ścieżki rozwiązanie jest wyświetlane oraz zapisywane do pliku tekstowego.

2. Opis algorytmu Bellmana-Forda

Jest to algorytm służący do wyszukiwania najkrótszych ścieżek w grafie ważonym z wierzchołka źródłowego do wszystkich pozostałych wierzchołków. Idea algorytmu opiera się na metodzie relaksacji. W odróżnieniu od algorytmu Dijkstry, algorytm Bellmana-Forda działa poprawnie także dla grafów z wagami ujemnymi (nie może jednak wystąpić cykl o łącznej ujemnej wadze osiągalny ze źródła. Posiada on jednak wyższą złożonością czasową. Złożoność czasowa oraz pamięciowa prezentują się następująco:

- Czasowa: $O(|V| \cdot |E|)$
- Pamięciowa: $O(|V|)$

Powyższy algorytm w pseudokodzie wygląda następująco:

```
Bellman-Ford(G,w,s):

dla każdego wierzchołka v w V[G] wykonaj
 d[v] = nieskończone
 poprzednik[v] = niezdefiniowane
d[s] = 0
dla i od 1 do |V[G]| - 1 wykonaj
  dla każdej krawędzi (u,v) w E[G] wykonaj
   jeżeli d[v] > d[u] + w(u,v) to
    d[v] = d[u] + w(u,v)
    poprzednik[v] = u
```

3. Zaimplementowane algorytmy:

• **struct lista_s** {
int v,w; /// zmienna v przechowuje nam sasiadow, a w wage
lista_s* nastepny;
}

• **void wyswietl_sciezke(int* poprzednik, int* koszt_dojscia, int wierzch)** – funkcja ma za zadanie wyświetlenie na standardowe wyjście ścieżki dojścia z wierzchołka startowego do pozostałych oraz ich kosztów dojść.

• **void wyswietl_lista(lista_s** tab_list, int wierzch)** – funkcja ta ma za zadanie wyświetlenie zawartości grafu w postaci reprezentacji listy sąsiedztwa na standardowe wyjście.

• **void wyswietl_macierz(int** graf_m, int wierzch)** – funkcja ta ma za zadanie wyświetlenie zawartości grafu w postaci reprezentacji macierzy sąsiedztwa na standardowe wyjście.

• **void wypisz_dane(int wierzcholki, int il_krawedz, fstream& strumien)** – funkcja wypisuje do pliku „dane.txt” grafy dla zadanych ilości wierzchołków oraz krawędzi tworząc najpierw główny cykl łączący wszystkie wierzchołki, a następnie dodaje względem ilości

krawędzi pozostałych do wygenerowania kolejne krawędzie. Funkcja również losuje wierzchołek startowy z przedziału <0 , wierzchołki>.

• **void generuj_dane(fstream& strumien)** – funkcja generuje ilości wierzchołków oraz krawędzi a następnie wywołuje po 100 razy dla każdego z przypadków funkcję wypisz dane. Funkcja wpierw tworzy i/lub opróżnić plik „dane.txt" z danych.

• **void pomiar(ifstream& strumien)** – funkcja ta otwiera nam strumień z prawem do odczytu danych dla pliku „dane_do_pliku.txt" oraz wywołuje funkcję znajdz_droge a następnie zamyka strumień.

• **void znajdz_droge(ifstream& wejscie)** – funkcja ta odpowiada za zainicjowanie strumienia danych wyjscie do pliki „pomiar.txt" z prawem do zapisu w wersji dopisywania danych oraz go zamknięcie go na sam koniec. Po otwarciu strumienia danych funkcja 2000 razy pobiera z wejscia (pliku „dane_do_pliku.txt") kolejno liczbę wierzchołków, krawędzi oraz numer wierzchołka startowego. Następnie funkcja alokuje pamięć dla wczytania tablicy dynamicznej dwuwymiarowej int** graf oraz kolejno uruchamia funkcje wczytaj_graf, znajdz_d_lista, znajdz_d_macierz i usuwa zajmowaną przez graf pamięć.

• **void znajdz_d_macierz(int** graf, int& wierzch, int& kraw, int& start, fstream& wyjscie)** – funkcja ta alokuje pamięć dla dynamicznej tablicy dwuwymiarowej int** graf_m a następnie wywołuje funkcję graf_macierz. Następnie zaczyna odmierzanie czasu poprzez zainicjowanie zmiennej czas1 oraz wywołuje algorytm wyznaczania najkrótszych dróg do wszystkich pozostałych wierzchołków dla danego wierzchołka startowego BF_macierz a następnie kończy odliczanie czasu inicjując zmienną czas2, od której odejmowany jest czas1 w celu obliczenia różnicy w czasie pomiędzy tymi zmiennymi. Następnie funkcja usuwa wcześniej przydzieloną pamięć dla graf_m i oblicza czas trwania algorytmu rzutując zmienną czas2 na double czas_trwania oraz dzieląc przez CLOCKS_PER_SEC. Na sam koniec funkcja na wyjscie („pomiar.txt") wypisuje czas trwania algorytmu dla macierzy sąsiedztwa.

• **void znajdz_d_lista(int** graf, int& wierzch, int& kraw, int& start, fstream& wyjscie)** – funkcja ta robi to samo co poprzednia z różnicą w tym, że tutaj reprezentacją grafu jest tablica list -> lista_s** tab_list oraz funkcje wywoływane kolejno to graf_lista, BF_lista przez co pamięć jest alokowana oraz usuwana w nieco inny sposób.

• **void** wczytaj_graf(int** graf, int& wierzch, int& kraw, int& start, ifstream& wejscie**)** – funkcja ta ma za zadanie wczytać do tablicy dynamicznej dwuwymiarowej int** graf kolejne krawędzie grafu. Dodatkowo wewnątrz tej funkcji zostały pozostawione zakomentowane linijki odpowiadające za wyświetlanie grafu na standardowe wyjście.

• **void** graf_macierz(int& wierzch, int& kraw, int& start, int** graf, int** graf_m)** – funkcja ta na początku zeruje wszystkie krawędzie, czyli ustawia ich „brak" dla całej macierzy, a następnie przepisuje kolejno każdą krawędź z int** graf do int** graf_m, który jest reprezentacją grafu w formie macierzy sąsiedztwa. Dodatkowo została zaimplementowana możliwość wyświetlenia macierzy sąsiedztwa na standardowe poprzez funkcję wyswietl_macierz poprzez odkomentowanie jej wywołania na końcu funkcji graf_macierz.

• **void graf_lista(int& wierzch, int& kraw, int& start, int** graf, lista_s** tab_list, lista_s* lista)** – funkcja ta na początku uzupełnia int** tab_list nullami a następnie wpisuje kolejne int* lista do tab_list. Int* lista są strukturami, które zawierają numery sąsiadów danego wierzchołka oraz wskaźnik na kolejnego. Dodatkowo została zaimplementowana możliwość wyświetlenia listy sąsiedztwa na standardowe poprzez funkcję wyswietl_lista poprzez odkomentowanie jej wywołania na końcu funkcji graf_lista.

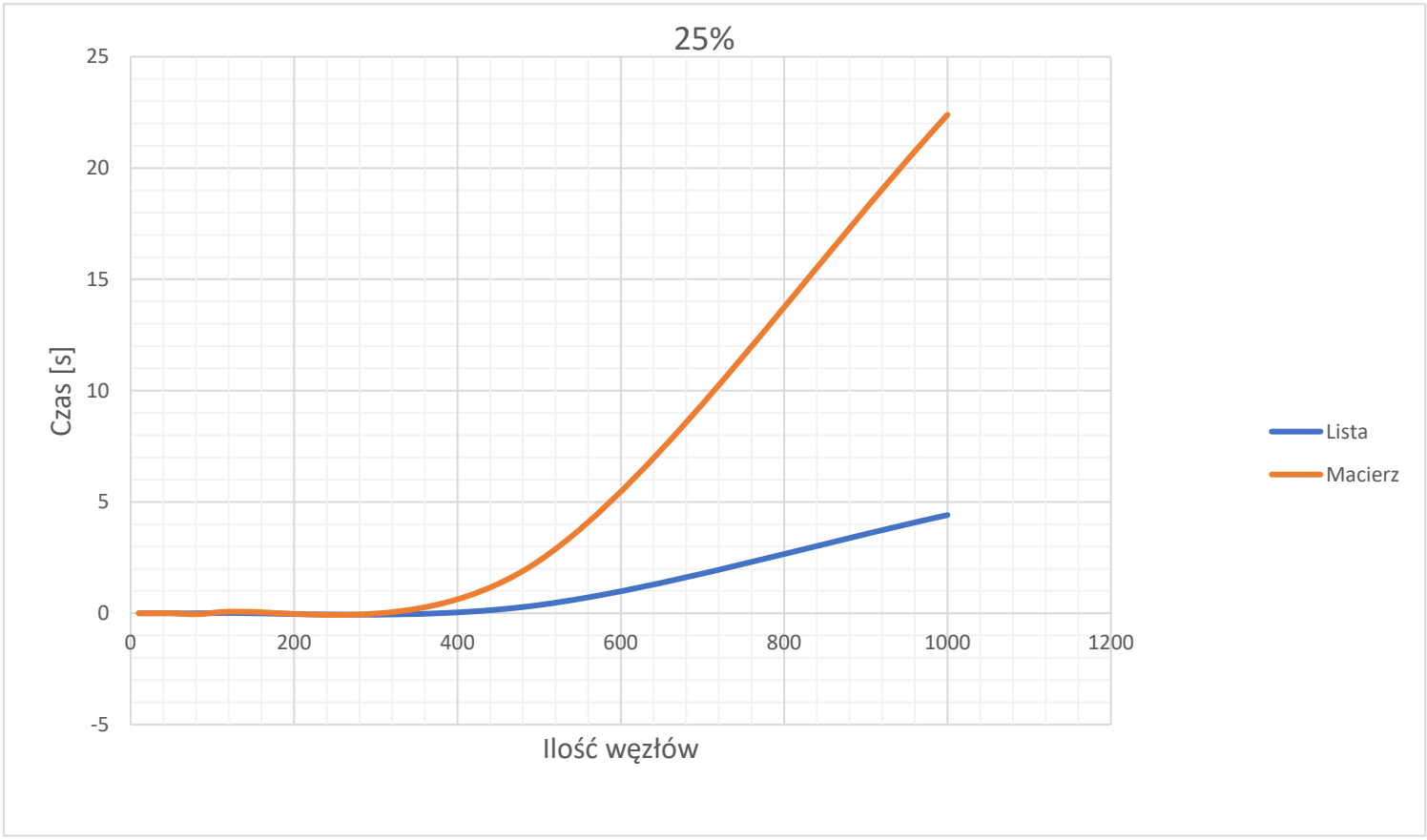
• **void BF_macierz(int** graf_m, int wierzch, int kraw, int start)** – funkcja ta na początku tworzy tablice dynamiczne i alokuje dla nich pamięć: int* koszt_dojscia oraz int* poprzednik. Następnie tablicę poprzednik wypełnia wartością -1, a tablicę koszt_dojscia wypełnia maksymalną wartością dla zmiennej typu int 32bitowej. Następnie uruchamiane jest przeszukiwanie macierzy sąsiedztwa algorytmem Bellmana-Forda poprzez relaksację kolejnych krawędzi. Dodatkowo została zaimplementowana możliwość wyświetlenia ścieżek na standardowe do wierzchołków poprzez funkcję wyswietl_sciezke poprzez odkomentowanie jej wywołania na końcu funkcji BF_macierz.

• **void BF_lista(int wierzch, int kraw, int start, lista_s** tab_list)** – funkcja ta działa tak samo jak jej poprzednik z tym, że zawiera wskaźnik lista_s* sasiedzi , który służy za wskaźnik wczytujący kolejne listy z tab_list oraz jest dostosowana pod reprezentację grafu w postaci listy sąsiedztwa. Dodatkowo została zaimplementowana możliwość wyświetlenia ścieżek na standardowe do wierzchołków poprzez funkcję wyswietl_sciezke poprzez odkomentowanie jej wywołania na końcu funkcji BF_lista.

4. Otrzymane wyniki:

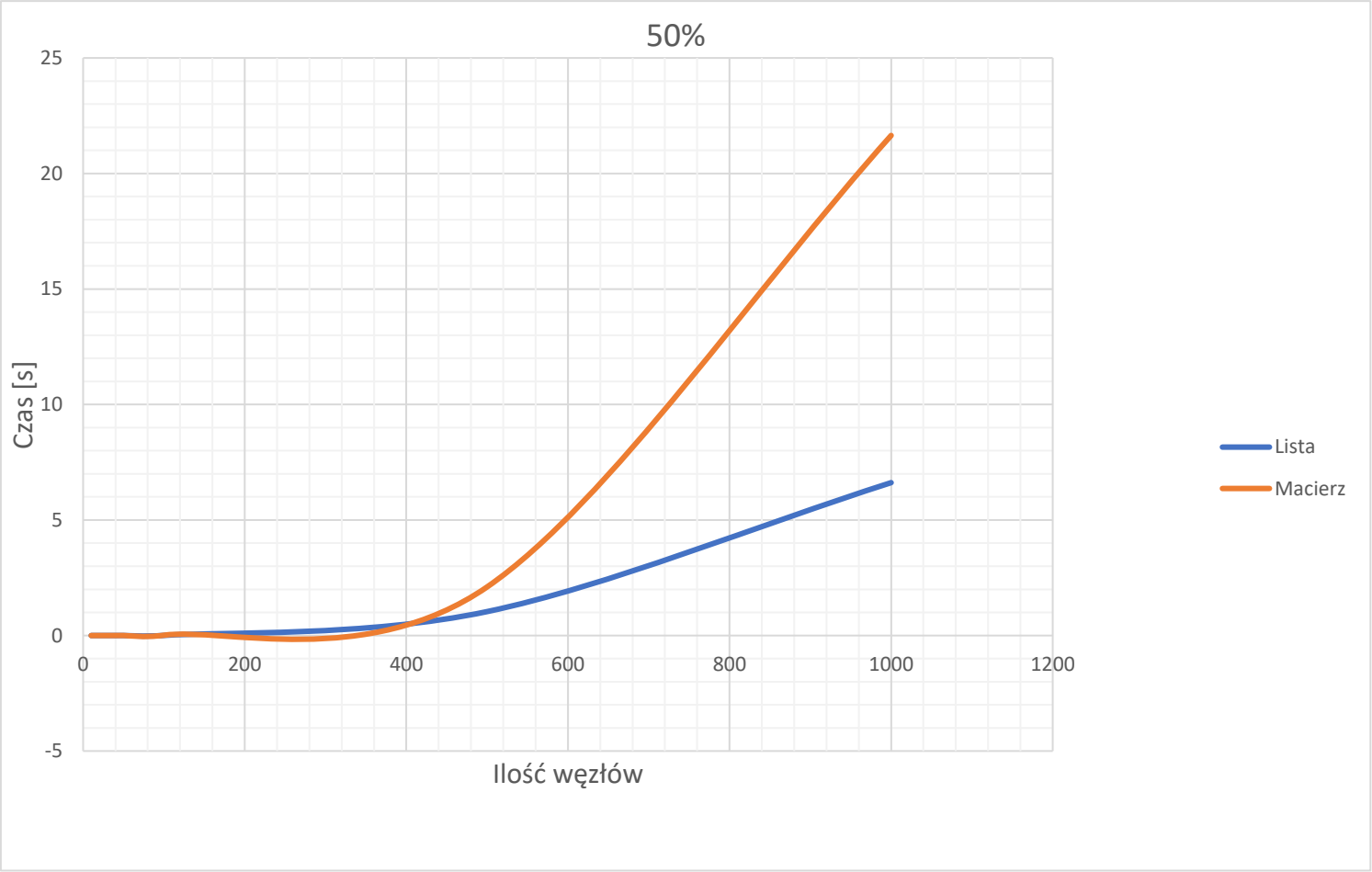
- **25%**

	10	50	100	500	1000
Lista	0,000019	0,000489	0,003149	0,367222	4,40937
Macierz	0,000079	0,001435	0,014373	2,35375	22,3921
Stosunek Lista/Macierz	0,240506	0,340766551	0,219091	0,156016	0,196916



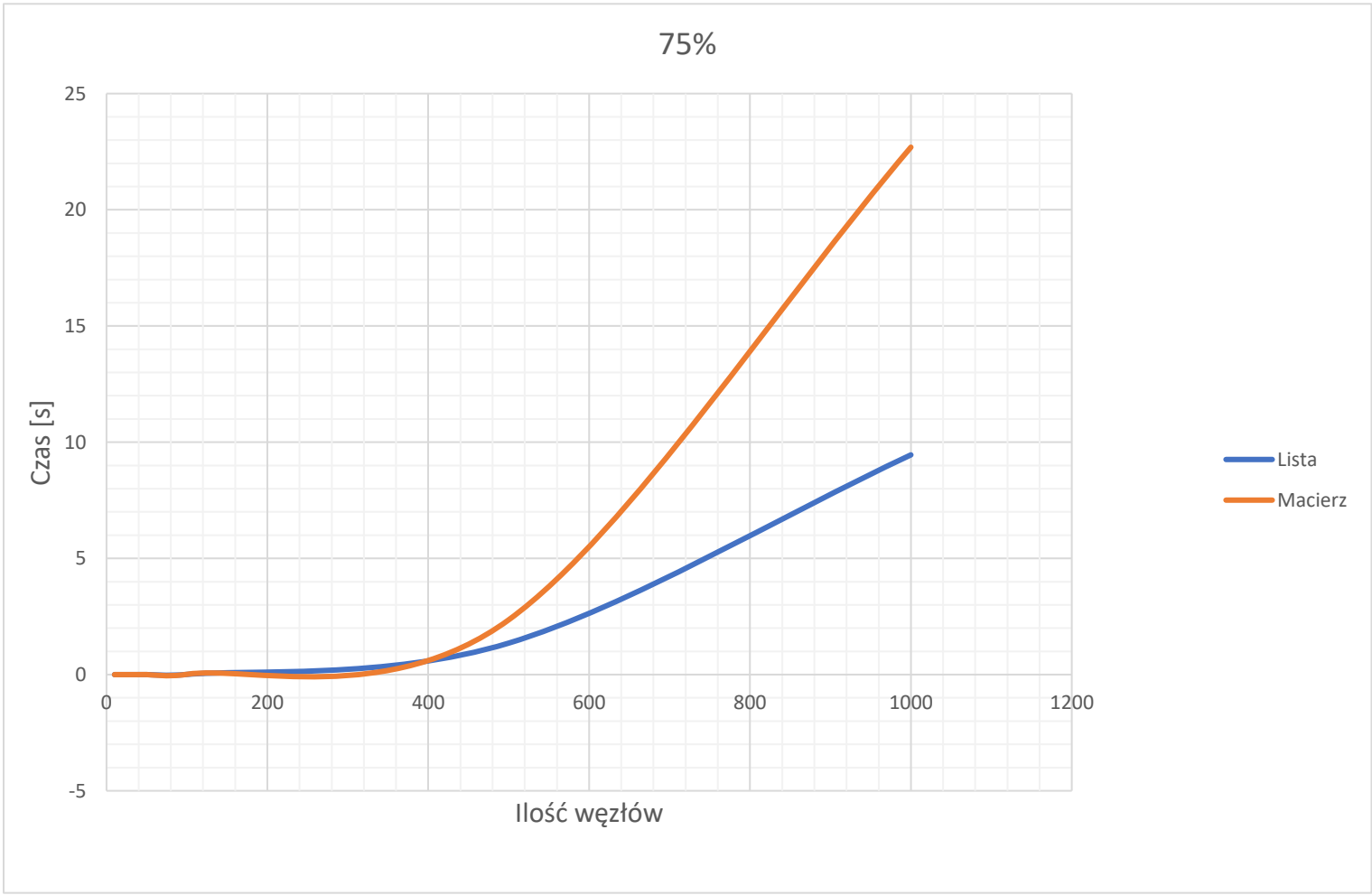
- **50%**

	10	50	100	500	1000
Lista	0,000026	0,000737	0,004855	1,03645	6,61695
Macierz	0,000077	0,001793	0,013233	2,1036	21,644
Stosunek Lista/Macierz	0,3376623	0,411042945	0,366886	0,492703	0,305718



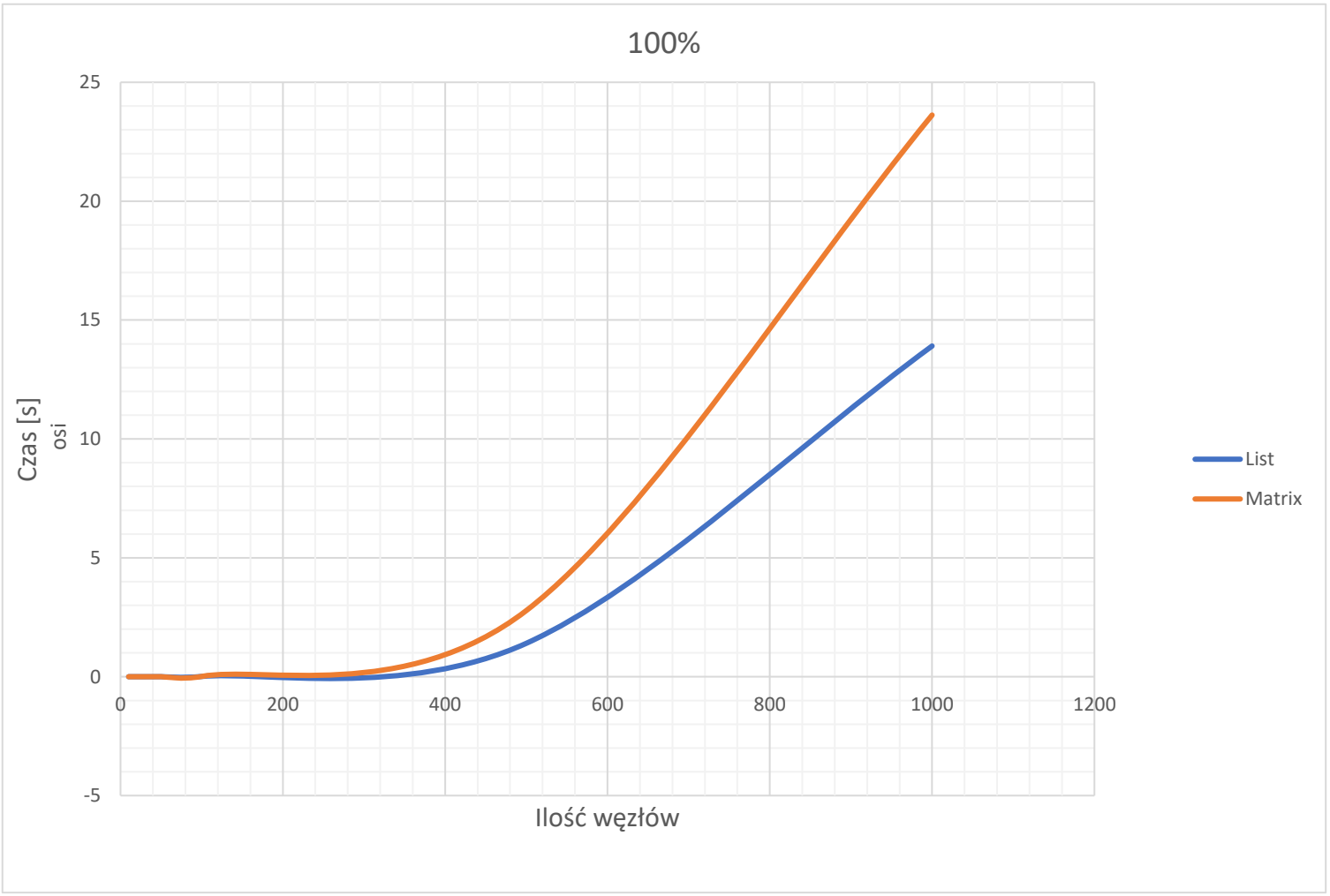
- **75%**

	10	50	100	500	1000
Lista	0,000032	0,001357	0,008599	1,35547	9,45505
Macierz	0,000082	0,002057	0,012803	2,35646	22,6937
Stosunek Lista/Macierz	0,390244	0,65969859	0,671639	0,575215	0,416638

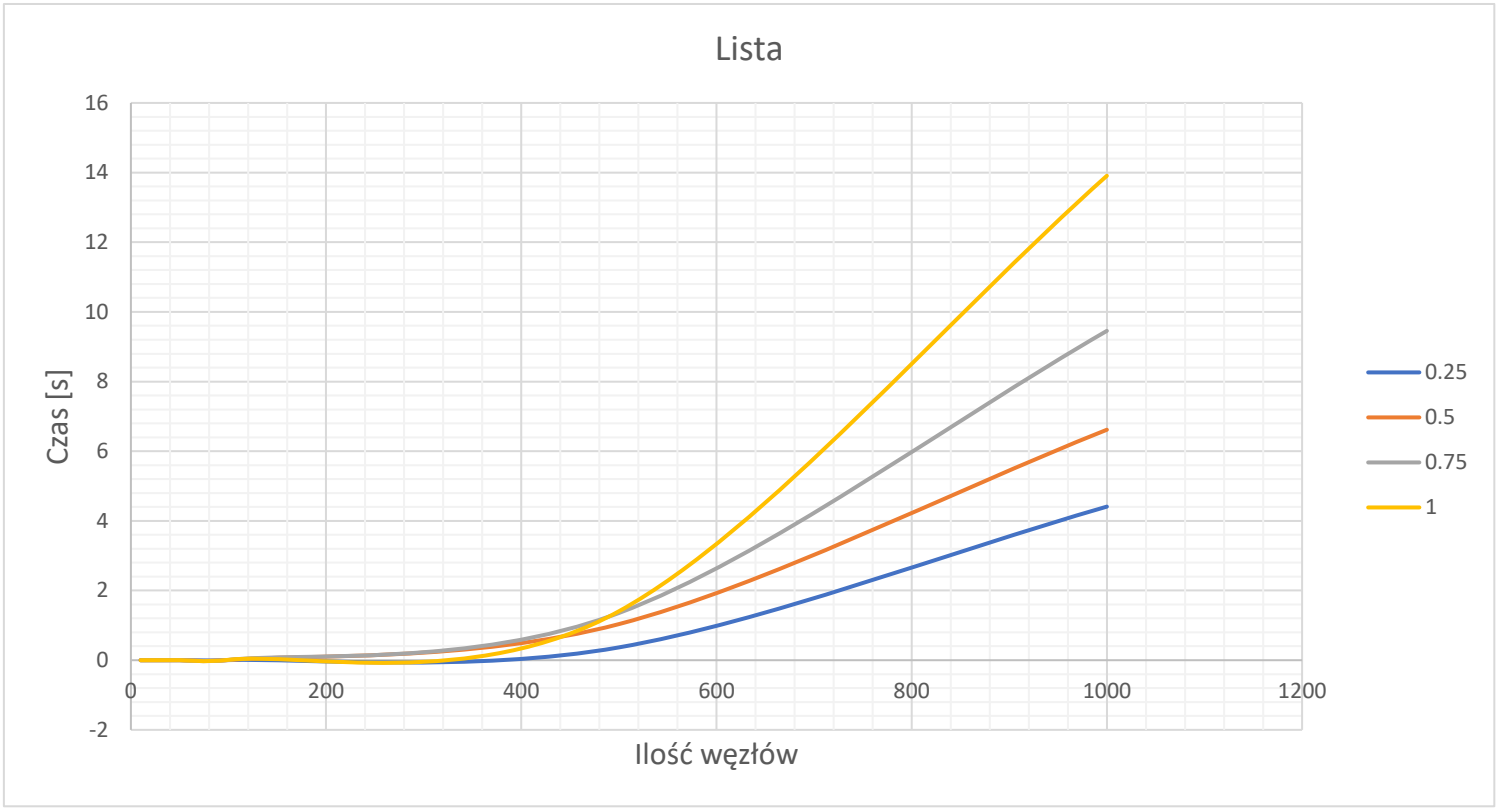


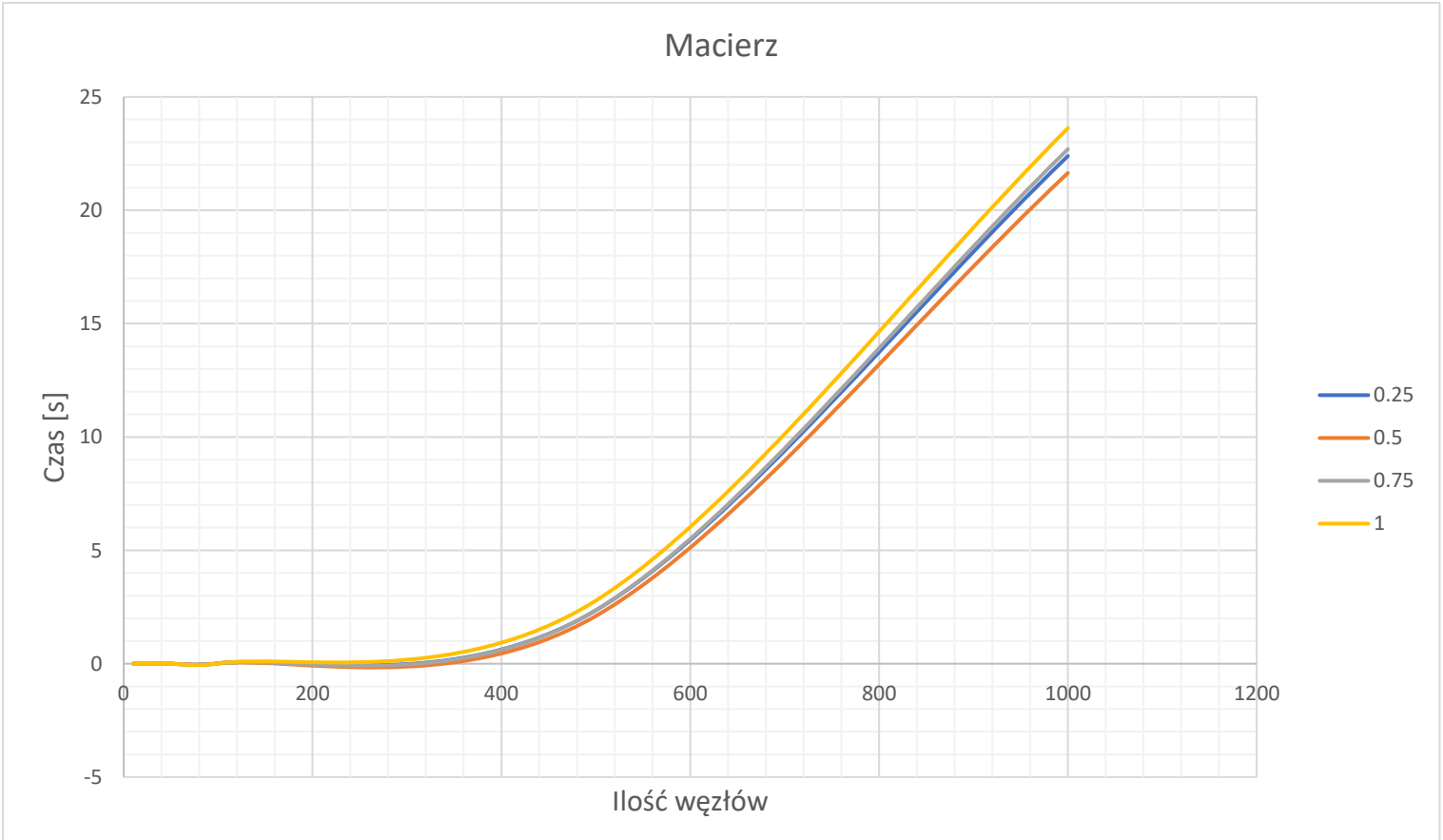
- **100%**

	10	50	100	500	1000
Lista	0,000041	0,001246	0,010298	1,40472	13,9079
Macierz	0,000078	0,001444	0,011383	2,78242	23,6229
Stosunek Lista/Macierz	0,525641	0,862880886	0,904682	0,504855	0,588747



- **Wykresy dla parametru w postaci typu reprezentacji:**





5. Wnioski

Osiągnięte rezultaty przedstawiają, że w większości przypadków dla reprezentacji grafu za pomocą listy sąsiedztwa wyniki są bardziej efektywne niż dla reprezentacji za pomocą macierzy sąsiedztwa. Może to być spowodowane tym, że przeszukiwanie całej tablicy jest wolniejsze niż dostęp do danych w zaimplementowanej liście. Kolejnym powodem może być fakt, że macierz potrzebuje więcej pamięci niż lista. Ponadto wraz ze wzrostem gęstości, czyli ilości krawędzi czas wykonywania dla reprezentacji w postaci listy, zwiększa się. Dla macierzy sąsiedztwa czas algorytmu Bellmana-Forda jest w przybliżeniu niezależny od gęstości grafu, delikatnie wzrasta ze wzrostem gęstości.