Projekt 2 - Grafy

Filip Sanowski

Projektowanie i Analiza Algorytmów

`Pełny kod źródłowy, w repozytorium GitHub`

1 Wstęp

Celem projektu było zaimplementowanie oraz porównanie efektywności wybranych algorytmów grafowych: przeszukiwania wszerz (BFS), w głąb (DFS), algorytmu Dijkstry oraz Bellmana-Forda. Algorytmy te zostały przetestowane na grafach skierowanych, reprezentowanych zarówno jako lista sąsiedztwa, jak i macierz sąsiedztwa.

2 Opis algorytmów

2.1 BFS (Wyszukiwanie wszerz)

Opis działania:
Algorytm BFS przeszukuje graf warstwami — zaczynając od wierzchołka startowego, odwiedza wszystkich jego sąsiadów, a następnie sąsiadów tych wierzchołków, itd. W implementacji wykorzystano kolejkę, która przechowuje wierzchołki do odwiedzenia. Każdy wierzchołek jest oznaczany jako odwiedzony w momencie dodania go do kolejki.

Złożoność czasowa:

- Lista sąsiedztwa: $O(V + E)$
- Macierz sąsiedztwa: $O(V^2)$

Użyte parametry:

- Użyto struktury `std::queue`
- Wektor `odwiedzony[]` do oznaczania odwiedzin

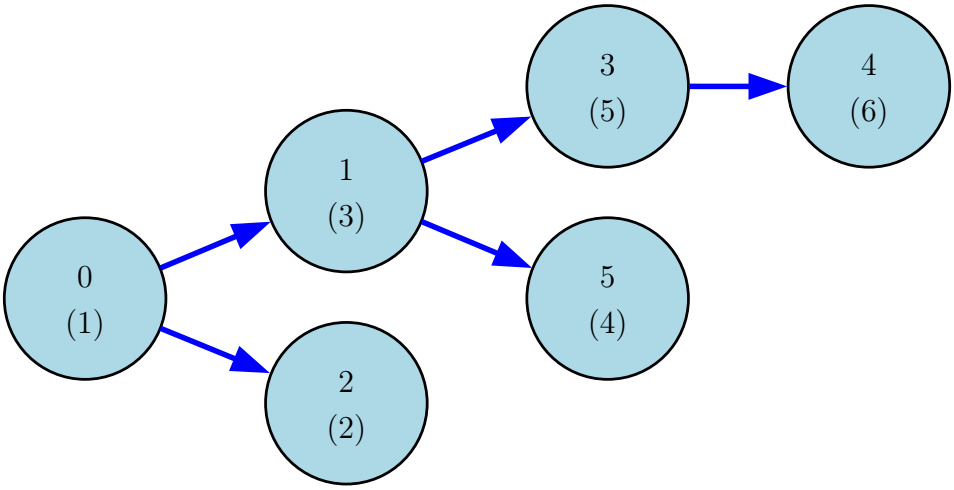
Zalety:

- Optymalna w wersji z listą sąsiedztwa

Wady:

- Nieefektywna w grafach z niską gęstością w wersji z macierzą

Wizualizacja:



Rysunek 1: Wizualizacja kolejności odwiedzania wierzchołków w BFS

2.2 DFS (Wyszukiwanie wgłąb)

Opis działania:
Algorytm DFS eksploruje graf w głąb — z każdego wierzchołka przechodzi do pierwszego nieodwiedzonego sąsiada, aż do momentu, gdy dalsze przejście jest niemożliwe. Następnie następuje powrót do poprzedniego wierzchołka i kontynuacja przeszukiwania.

Złożoność czasowa:

- Lista sąsiedztwa: $O(V + E)$
- Macierz sąsiedztwa: $O(V^2)$

Użyte parametry:

- Użyto struktury `std::stack`

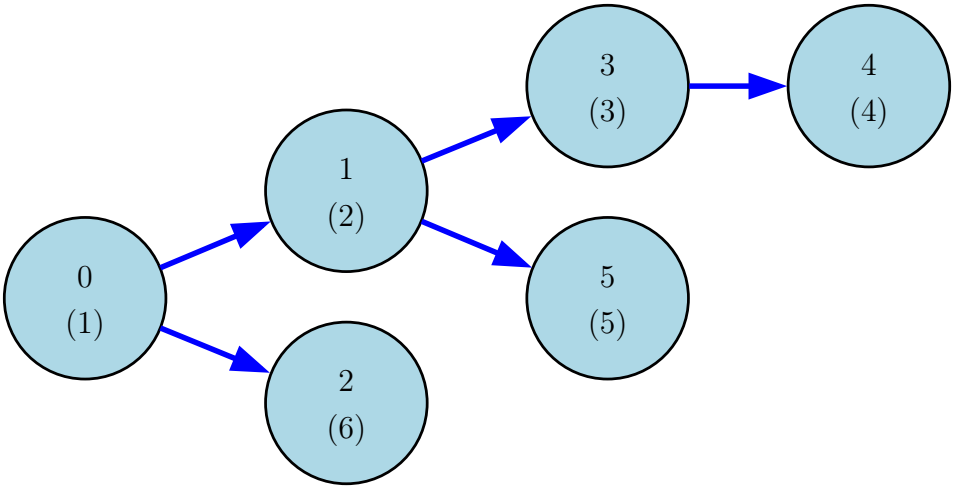
Zalety:

- Optymalna w wersji z listą sąsiedztwa

Wady:

- Nieefektywna w grafach z niską gęstością w wersji z macierzą

Wizualizacja:



Rysunek 2: Wizualizacja kolejności odwiedzania wierzchołków w DFS

2.3 Algorytm Dijkstry

Opis działania:
Algorytm Dijkstry znajduje najkrótsze ścieżki z jednego wierzchołka do wszystkich innych w grafie o nieujemnych wagach. W każdej iteracji wybierany jest nieodwiedzony wierzchołek o najmniejszym dystansie. Następnie następuje relaksacja wszystkich krawędzi wychodzących z tego wierzchołka.

Złożoność czasowa:

- Lista sąsiedztwa: $O(V^2 + E)$
- Macierz sąsiedztwa: $O(V^2)$

Użyte parametry:

- Użyto wektora `dystans[]` i `odwiedzony[]`

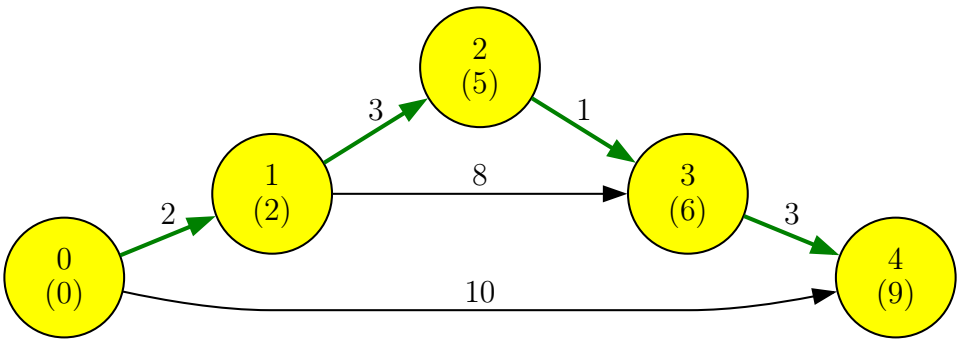
Zalety:

- Znajduje dokładne najkrótsze ścieżki przy dodatnich wagach
- Wersja z listą oszczędza czas przeglądania sąsiadów

Wady:

- Nie działa poprawnie przy ujemnych wagach
- W wersji macierzowej nieefektywny dla dużych grafów z niskim zagęszczeniem

Wizualizacja:



Rysunek 3: Wynik końcowy po zastosowaniu algorytmu Dijkstry (wyliczenie najkrótszej możliwej trasy od punktu startowego dla każdego wierzchołka)

2.4 Algorytm Bellmana-Forda

Opis działania:
Algorytm Bellmana-Forda służy do wyznaczania najkrótszych ścieżek z jednego źródła w grafie z krawędziami o dowolnych wagach, w tym ujemnych. Działa poprzez $V-1$ iteracji relaksacji wszystkich krawędzi. Każde zmniejszenie dystansu skutkuje dalszymi próbami relaksacji.

Złożoność czasowa:

- Lista sąsiedztwa: $O(V \cdot E)$
- Macierz sąsiedztwa: $O(V^3)$

Użyte parametry:

- użuto wektora `dystans[]` do przechowywania długości najkrótszych ścieżek
- Implementacja wykrywa cykl ujemny

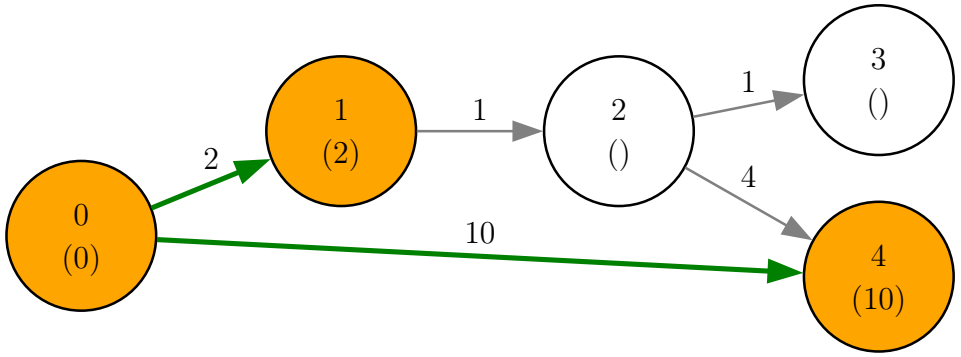
Zalety:

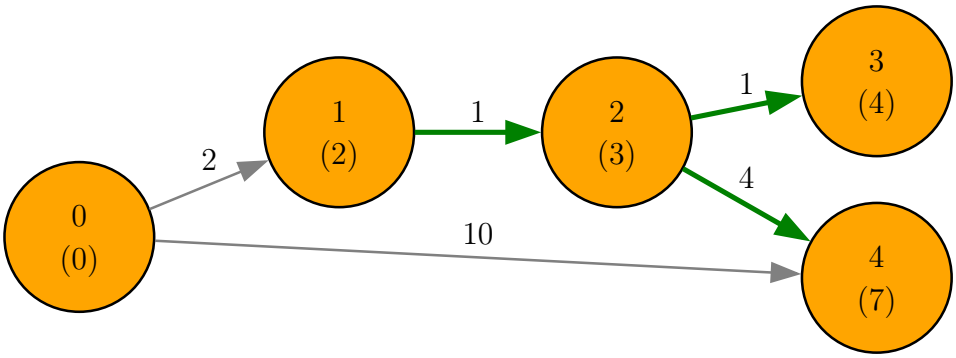
- Obsługuje grafy z ujemnymi wagami
- Wykrywa istnienie cykli o ujemnej wadze

Wady:

- Znacznie wolniejszy od Dijkstry w przypadku dużych grafów
- W wersji z macierzą bardzo nieefektywny ($O(V^3)$)

Wizualizacja:





Rysunek 4: Wizualizacja stopniowego procesu relaksacji w algorytmie Belmana-Forda

2.5 Implementacja algorytmów

Kod z implementacją algorytmów dla wersji z listą sąsiedztwa
Kod z implementacją algorytmów dla wersji z macierzą sąsiedztwa

3 Analiza i weryfikacja algorytmów

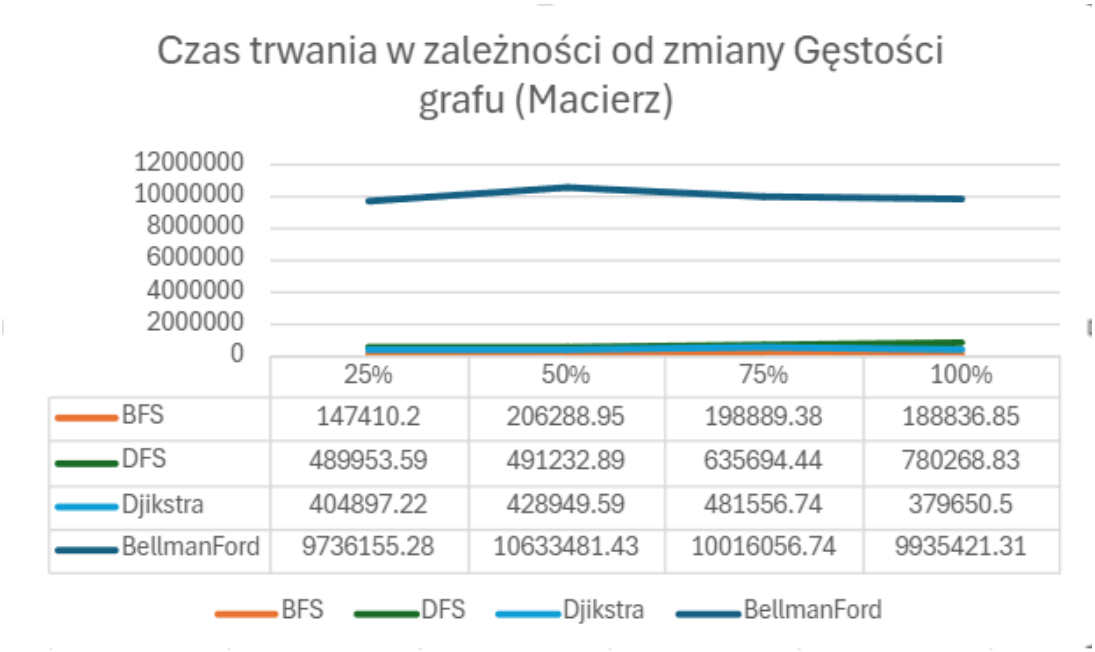
W celu oceny efektywności zaimplementowanych algorytmów grafowych przeprowadzono szereg testów czasowych na losowo generowanych grafach skierowanych. Testy wykonano dla różnych rozmiarów grafu (10, 50, 100, 200, 500) oraz różnych gęstości krawędzi (25%, 50%, 75%, 100%).

Parametry testów

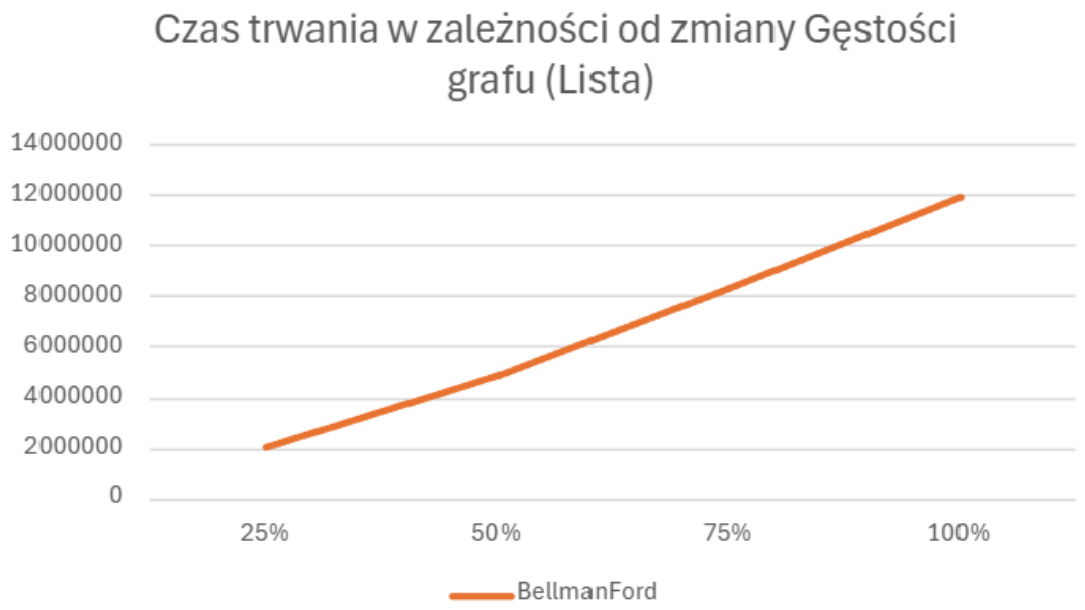
Dla każdego algorytmu wykonano po 100 powtórzeń dla danego rozmiaru grafu i gęstości. Uśrednione wyniki zapisano do plików `.csv`, a następnie wygenerowano wykresy w programie Excel. Pomiar czasu został zrealizowany z użyciem biblioteki `<chrono>` w C++, z dokładnością do nanosekundy.

Zmieniane parametry wejściowe

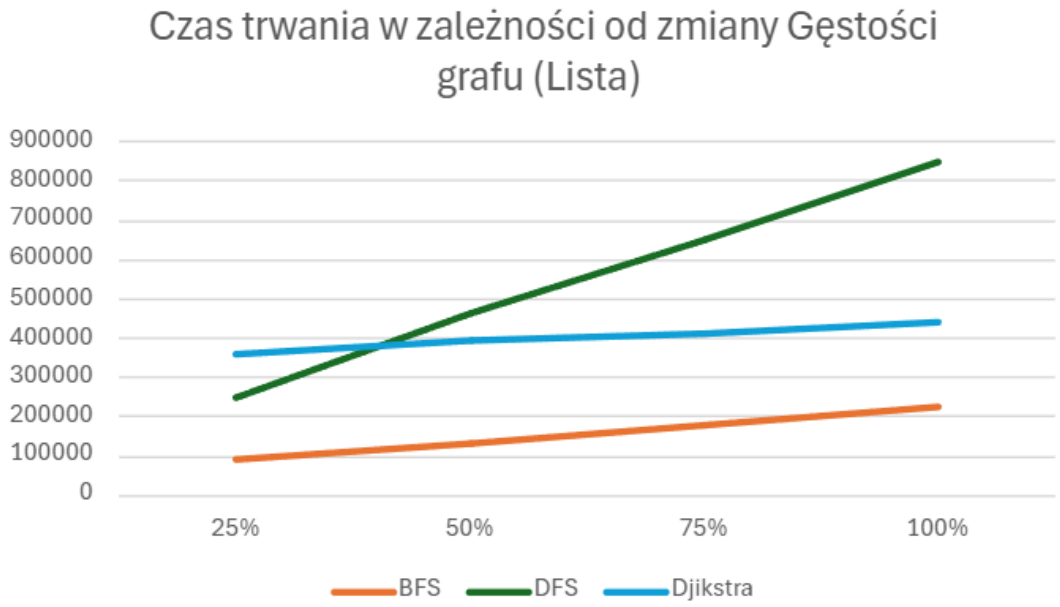
- Liczba wierzchołków $V \in \{10, 50, 100, 500, 1000\}$
- Gęstość grafu $d \in \{10\%, 25\%, 50\%, 75\%, 100\%\}$
- Wagi krawędzi: losowe z zakresu $[1, 10]$



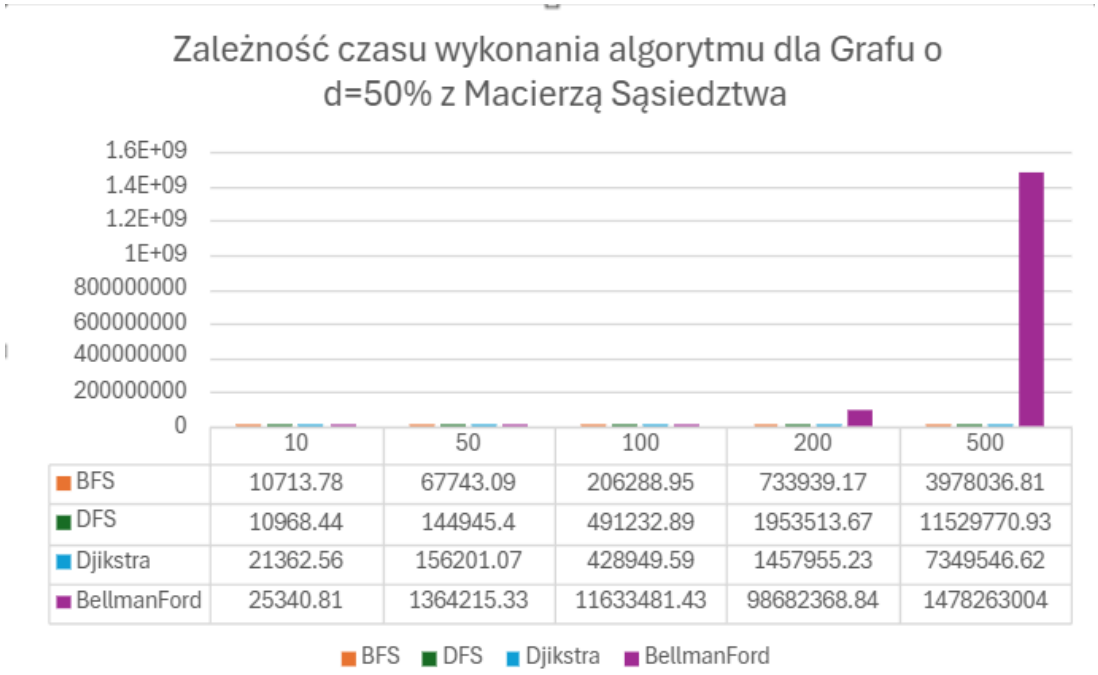
Rysunek 5: Zależność czasu działania algorytmów od gęstości grafu (macierz sąsiedztwa)



Rysunek 6: Zależność czasu działania Bellmana-Forda od gęstości grafu (lista sąsiedztwa)

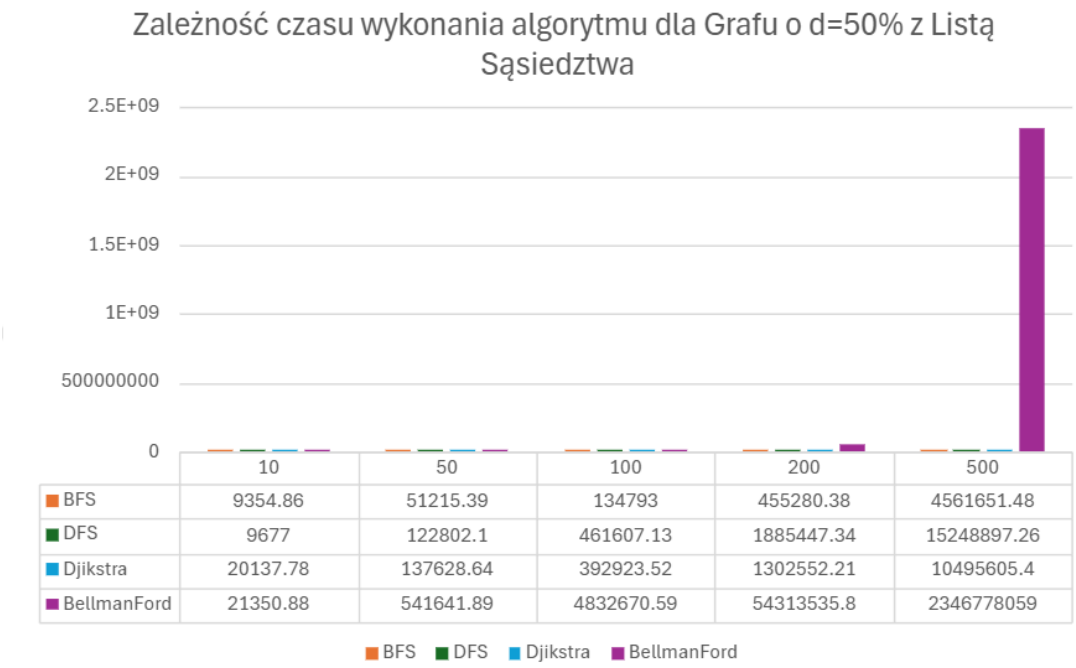


Rysunek 7: Zależność czasu działania BFS, DFS i Dijkstry od gęstości grafu (lista sąsiedztwa)



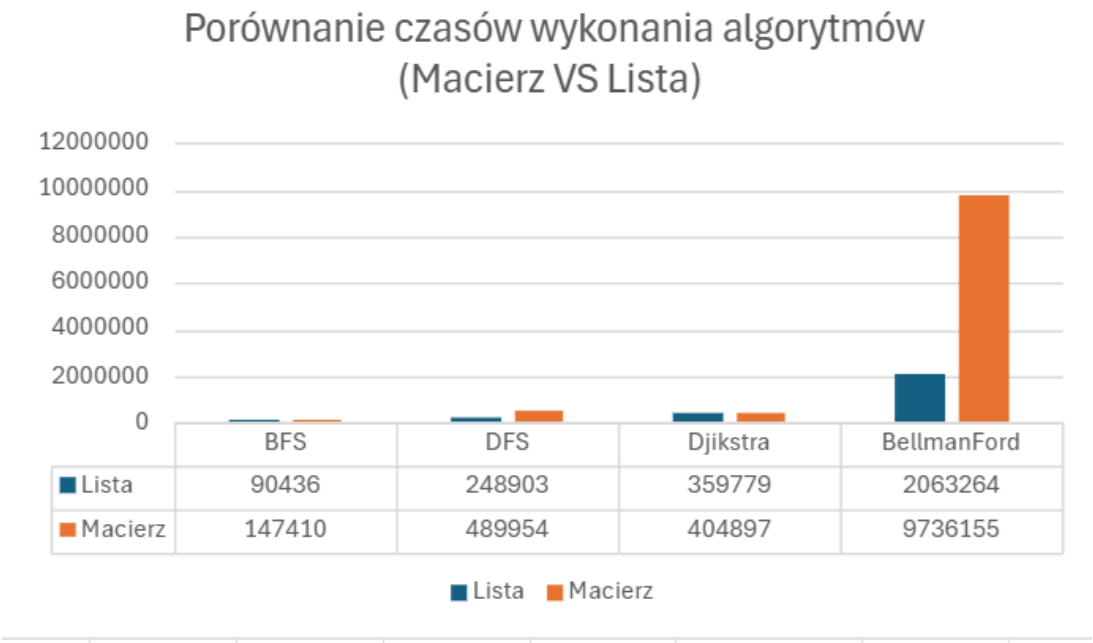
	10	50	100	200	500
BFS	10713.78	67743.09	206288.95	733939.17	3978036.81
DFS	10968.44	144945.4	491232.89	1953513.67	11529770.93
Djikstra	21362.56	156201.07	428949.59	1457955.23	7349546.62
BellmanFord	25340.81	1364215.33	11633481.43	98682368.84	1478263004

Rysunek 8: Zależność czasu działania algorytmów od liczby wierzchołków (macierz sąsiedztwa, gęstość 50%)



	10	50	100	200	500
BFS	9354.86	51215.39	134793	455280.38	4561651.48
DFS	9677	122802.1	461607.13	1885447.34	15248897.26
Djikstra	20137.78	137628.64	392923.52	1302552.21	10495605.4
BellmanFord	21350.88	541641.89	4832670.59	54313535.8	2346778059

Rysunek 9: Zależność czasu działania algorytmów od liczby wierzchołków (lista sąsiedztwa, gęstość 50%)



	BFS	DFS	Djikstra	BellmanFord
Lista	90436	248903	359779	2063264
Macierz	147410	489954	404897	9736155

Rysunek 10: Porównanie czasów działania algorytmów dla listy i macierzy sąsiedztwa (V=100, gęstość 25%)

Interpretacja wykresów

- **Wpływ gęstości na czas Trwania algorytmów**
  Możemy zaobserwować że zmiana gęstości nie za bardzo wpływa na algorytmy oparte

na strukturze z macierzą ponieważ ma ona zgóry ustalony rozmiar ale wpływa znacząco na grafy z listą sąsiedztwa ponieważ każdy przyrost krawędzi powoduje zwiększenie iteracji w ich algorytmach.

- **Zależność czasu działania od liczby wierzchołków**
  Wykresy te pokazują wpływ rozmiaru grafu na czas wykonania dla obu reprezentacji. Zarówno dla listy jak i macierzy sąsiedztwa wzrost dla BFS i DFS jest łagodniejszy. Dijkstra rośnie szybciej, a Bellman-Ford wykazuje silny wzrost. W przypadku macierzy wzrost czasu działania jest znacznie szybszy — dla Bellmana-Forda wręcz wykładniczy — co potwierdza nieefektywność tej reprezentacji dla dużych grafów.

- **Porównanie czasu działania dla różnych struktur danych**
  Wykres przedstawia zestawienie średniego czasu działania każdego z czterech algorytmów dla tej samej liczby wierzchołków i gęstości (np. $V = 100$, gęstość 25%), w dwóch wariantach struktury: lista sąsiedztwa i macierz sąsiedztwa. Widoczna jest istotna różnica na korzyść listy dla większości przypadków macierz sąsiedztwa jest wolniejsza (ale dalej moim zdaniem prostsza do implementacji)

- **Porównanie czasu działania między Algorytmem Djikstra a Algorytmem Bellmana-Forda**
  Możemy zaobserwować że w mojej implementacji algorytm Bellmana-Forda jest o wiele wolniejszy od algorytmu Djikstry, ponieważ ma on z góry ustaloną ilość iteracji (często większą niż potrzebna)

Podsumowując, reprezentacja grafu ma istotne znaczenie dla czasu działania algorytmów — szczególnie w przypadku dużych i nie gęstych grafów, gdzie stanowczo wygrywa lista sąsiedztwa, zwłaszcza dla Bellmana-Forda i Dijkstry, których złożoność silnie zależy od liczby przetwarzanych krawędzi, a nie tak jak w wersji z macierzą gdzie ta złożoność jest sztywnie określona przez rozmiar macierzy.

4 Wnioski

4.1 Zgodność z teoretyczną złożonością

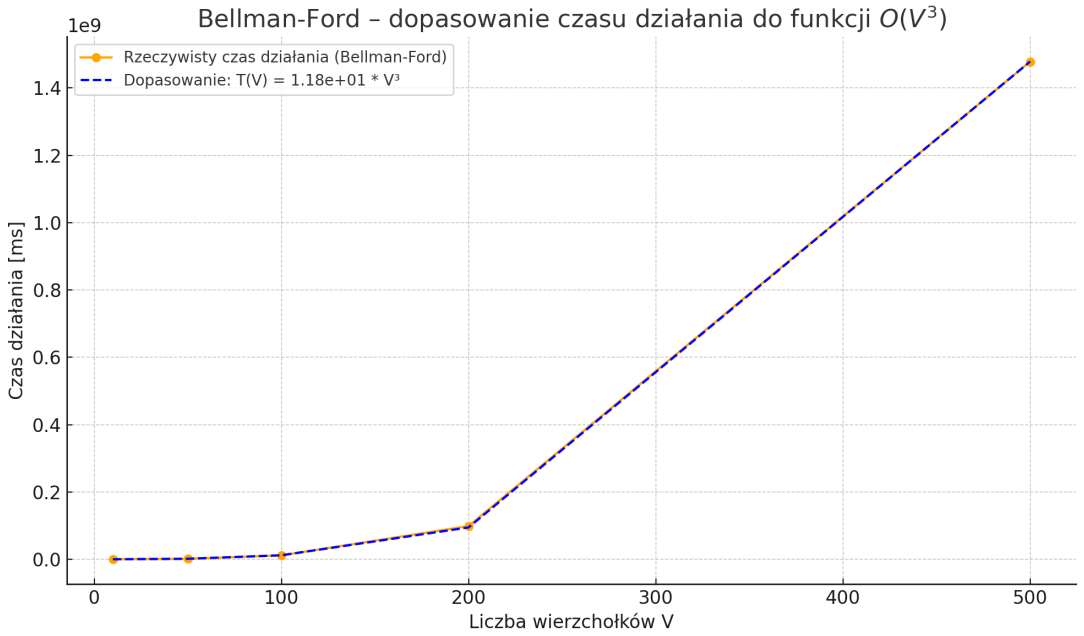
Aby zweryfikować zgodność czasów działania algorytmów z ich teoretyczną złożonością obliczeniową, przeprowadzono dopasowanie regresji dla funkcji złożoności:

- $T(V) = a \cdot V^3$ dla algorytmu Bellmana-Forda w wersji z macierzą sąsiedztwa
- $T(V) = a \cdot V^2$ dla algorytmu Dijkstry w wersji z listą sąsiedztwa

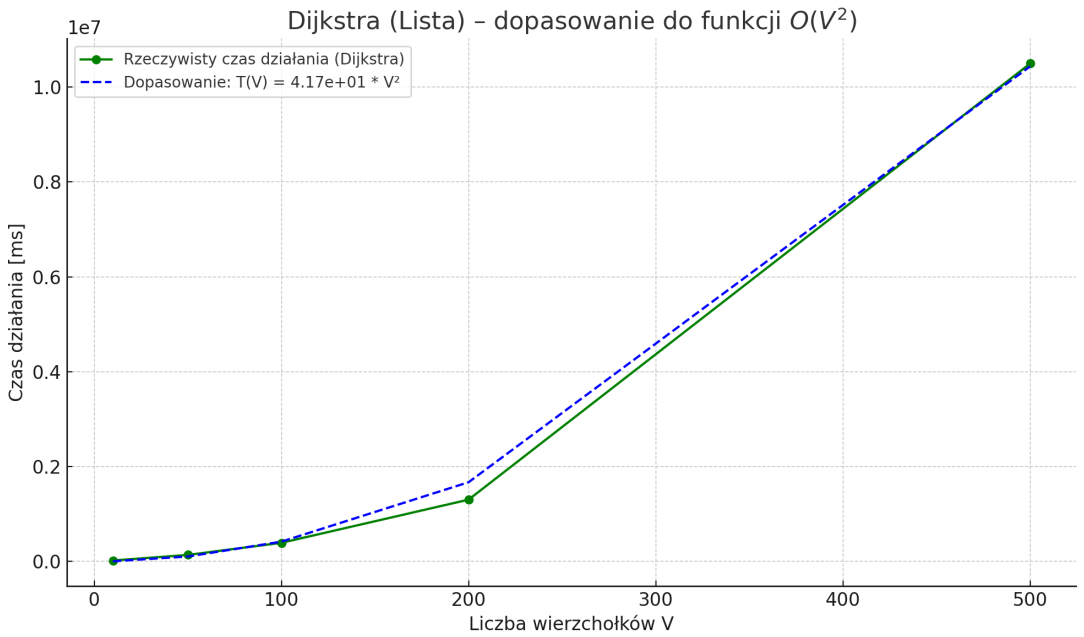
W obu przypadkach dopasowanie zostało wykonane metodą najmniejszych kwadratów. Uzyskano następujące współczynniki a:

- Bellman-Ford (macierz): $a \approx 11.83$
- Dijkstra (lista): $a \approx 41.74$

Dopasowanie wizualnie potwierdza poprawność złożoności teoretycznej — przebieg czasów rzeczywistych jest zgodny z funkcjami wzrostu $O(V^2)$ i $O(V^3)$.



Rysunek 11: Dopasowanie czasu działania algorytmu Bellmana-Forda do funkcji $O(V^3)$ (macierz)



Rysunek 12: Dopasowanie czasu działania algorytmu Dijkstry do funkcji $O(V^2)$ (lista)

Trudności w realizacji projektu

- Oddzielenie implementacji dla listy i macierzy przy zachowaniu tej samej logiki działania

- Stworzenie poprawnej struktury grafu z listą sąsiedztwa
- Obliczenie teoretycznej złożoności czasowej jak i badanie jej poprawności
- Utrzymanie spójności interfejsów dla benchmarków i algorytmów

Ciekawe obserwacje

- Lista sąsiedztwa wymaga bardziej rozbudowanej struktury danych, ale zapewnia lepszą skalowalność – szczególnie widoczne w algorytmach, które iterują tylko po rzeczywistych sąsiadach.
- Wersje algorytmów oparte na macierzy sąsiedztwa są znacznie mniej wydajne dla dużych grafów, szczególnie przy niskiej gęstości ale są o wiele łatwiejsze do zaimplementowania.
- Bellman-Ford, mimo największej złożoności, w odróżnieniu od Djikstry jest zdolny do pracy z krawędziami ujemnymi — potencjalnie przydatny np. w analizie kosztów.