

Struktury Danych i Złożoność Obliczeniowa - Projekt: Sprawozdanie

Wydział Elektroniki

Kierunek: Informatyka Techniczna

Grupa zajęciowa Wt 15:15

Semestr: 2020/2021 Lato

Prowadzący:

Dr inż. Dariusz Banasiak

Autor

Byczko Maciej 252747

Wstęp

Zadanie projektowe było napisanie programu i zmierzenie czasu wykonywania działań na:

- Wyznaczaniu minimalnego drzewa rozpinającego za pomocą:
 - Algorytmu Prima
 - Algorytmu Kruskala
- Najkrótszą ścieżkę w grafie:
 - Algorytm Dijkstry
 - Algorytm Bellmana-Forda

Powyższe algorytmy były wykonywane na następujących strukturach:

- Macierz sąsiedztwa - macierz o wymiarze $V \times V$ (V - Liczba wierzchołków), w której wiersz oznacza wierzchołek początkowy, kolumna wierzchołek końcowy a w punkcie wspólnym znajduje się waga krawędzi, jeżeli w punkcie wspólnym występuje 0, oznacza to że nie ma połączenia z danymi wierzchołkami.
- Lista sąsiedztwa - graf zapisany jako V list, gdzie w każdej liście są przechowywane numery wierzchołków do których można się dostać z danego wierzchołka (przykładowo na liście 2 przechowywane są wierzchołki do których można się dostać z wierzchołka 2)

Założenia

- wszystkie struktury danych powinny być alokowane dynamicznie,
- przepustowość/koszt krawędzi jest liczbą całkowitą (dodatnią),
- po zimplementowaniu każdego z algorytmów dla obu reprezentacji grafu należy dokonać pomiaru czasu działania algorytmów w zależności od rozmiaru grafu oraz jego gęstości.
- Badania należy wykonać dla 5 różnych (reprezentatywnych) liczb wierzchołków V oraz następujących gęstości grafu: 25%, 50%, 75% oraz 99%. Dla każdego zestawu: reprezentacja, liczba wierzchołków i gęstość należy wygenerować serię losowych instancji (np. 20, 50 lub 100 w zależności od rozrzułu poszczególnych wyników), zaś w sprawozdaniu umieścić wyłącznie wyniki uśrednione z danej serii,
- program powinien umożliwić sprawdzenie poprawności zaimplementowanych operacji i zbudowanej struktury grafu (dodatkowe informacje w dalszej części dokumentu),
- językami programowania są języki kompilowane do kodu natywnego (C, Objective C, C++, Rust, GO)
- implementacja projektu powinna być wykonana w formie jednego programu,

- nie wolno korzystać z gotowych bibliotek np. STL, Boost lub innych – wszystkie algorytmy i struktury muszą być zaimplementowane przez studenta
- kod źródłowy powinien być komentowany.
- realizacja zadania powinna być wykonana w formie jednego programu
- kod źródłowy powinien być komentowany
- program musi skompilowany do wersji exe

Dodatkowe informacje

- utworzenie struktury na podstawie danych zapisanych w pliku tekstowym. Pierwsza liczba określa rozmiar struktury, następnie należy wprowadzić odpowiednią liczbę danych np. każda liczba w osobnej linii
- wyświetlenie struktury na ekranie
- możliwość wykonania wszystkich przewidzianych operacji na danej strukturze
- Projekt został napisany w języku C++ w standardzie C++20
- Do pisania oraz komplikacji zostało użyte środowisko CLion
- Wykresy oraz dane zostały przetworzone za pomocą skryptu napisanego w Pythonie w wersji 3.9.4
- Losowe dane do losowych struktur są generowane za pomocą biblioteki `random`
- Wyniki zostały uśrednione za pomocą biblioteki `statistics` w Pythonie

Złożoność algorytmów

Reprezentacja za pomocą listy następników

Algorytm	Złożoność czasowa
Prim	$\$O(E / \log V)$
Kruskal	$\$O(E / \log E)$
Dijkstra	$\$O(E / \log V)$
Bellman-Ford	$\$O(E / \cdot V)$

Reprezentacja za pomocą macierzy sąsiedztwa

Algorytm	Złożoność czasowa
Prim	$\$O(V^2)$
Kruskal	$\$O(E / \log E)$
Dijkstra	$\$O(E / \log V)$
Bellman-Ford	$\$O(E / \cdot V)$

Plan eksperymentu

Informacje ogólne

- Pomiar czasu podczas wykonywania algorytmów dla następujących parametrów grafów:

- Rozmiary grafów od 10 do 100 wierzchołków (zwiększone co 10)
- Badane gęstości: 25%, 50%, 75%, 99%
- Funkcja mierząca czas: `std::chrono::high_resolution_clock`
- Sposób generacji struktur:
 - Tworzenie struktury z losowymi danymi o podanym rozmiarze (każdy pomiar miał generowaną losową strukturę zadanymi parametrami)
 - Wykonanie operacji mierzonej
 - Zapisanie wyniku do pliku
 - Powtórzenie operacji (wartość zadana przez użytkownika)

Generowanie grafu

Dla każdego typu algorytmu był wybierany odpowiedni typ tworzonego grafu:

- Graf nieskierowany dla algorytmu Prima oraz Kruskala:
 1. Stworzenie listy wszystkich wierzchołków
 2. Stworzenie listy wszystkich możliwych krawędzi (będzie to potrzebne w późniejszym etapie)
 3. Wylosowanie dwóch wierzchołków
 4. Utworzenie pomiędzy nimi połączenia nieskierowanego (dwustronnego) z losową wartością
 5. Usunięcie pierwszego wylosowanego wierzchołka
 6. Gdy zostaje spełniony warunek grafu spójnego (wszystkie wierzchołki są połączone) to następuje etap uzupełnienia gęstości grafu:
 1. Wylosowanie krawędzi z listy wszystkich możliwych krawędzi
 2. Utworzenie połączenia aby zastosować wylosowaną krawędź
 3. Usunięcie utworzonego połączenia z listy możliwych krawędzi
 4. Powtórzenie powyższych punktów dopóki nie zostanie spełniona gęstość grafu
- Graf skierowany dla algorytmu Dijkstry oraz Bellmana-Forda:
 1. Stworzenie listy wszystkich wierzchołków
 2. Stworzenie listy wszystkich możliwych krawędzi (będzie to potrzebne w późniejszym etapie)
 3. Utworzenie wierzchołka początkowego = 0
 4. Wylosowanie wierzchołka z wszystkich wierzchołków
 5. Utworzenie pomiędzy wierzchołkiem 0 oraz wierzchołkiem wylosowanym połączenia skierowanego z losową wartością
 6. Ustawienie wylosowanego wierzchołka na miejsce wierzchołka początkowego (na początku 0)
 7. Usunięcie wylosowanego wierzchołka z listy wszystkich wierzchołków
 8. Powtórzenie punktów 4 - 7 aż nie otrzymamy grafu spójnego
 9. Gdy zostaje spełniony warunek grafu spójnego (wszystkie wierzchołki są połączone) to następuje etap uzupełnienia gęstości grafu:
 1. Wylosowanie krawędzi z listy wszystkich możliwych krawędzi
 2. Utworzenie połączenia aby zastosować wylosowaną krawędź
 3. Usunięcie utworzonego połączenia z listy możliwych krawędzi

4. Powtórzenie powyższych punktów dopóki nie zostanie spełniona gęstość grafu

Każda krawędź ma nadaną wartość losową z przedziału od 1 do 999.

Pomiary czasowe

Pomiar czasowe były mierzone w **nanosekundach** \$(1 [ns] = 1 * 10^9 [s])\$ za pomocą następującej funkcji:

```
template<typename T>
double Timer(T i) {
    auto start = chrono::high_resolution_clock::now(); // Start the counter
    i(); // our function
    auto end = chrono::high_resolution_clock::now(); // Get value after executing
    function
    auto duration = end - start; // get time difference
    auto elapsed_time = chrono::duration_cast<chrono::nanoseconds>
    (duration).count(); // calculate time
    return elapsed_time; // Return executing time in nanoseconds
}
```

Wyniki wykonanych eksperymentów

Minimalne drzewo rozpinające (MST)

Opis ogólny Algorytmu Prima

Mając do dyspozycji graf nieskierowany i spójny, tzn. taki w którym krawędzie grafu nie mają ustalonego kierunku oraz dla każdych dwóch wierzchołków grafu istnieje droga pomiędzy nimi, algorytm oblicza podzbiór E' zbioru krawędzi E , dla którego graf nadal pozostaje spójny, ale suma kosztów wszystkich krawędzi zbioru E' jest najmniejsza możliwa

Schemat działania:

- Utwórz drzewo zawierające jeden wierzchołek, dowolnie wybrany z grafu.
- Utwórz kolejkę priorytetową, zawierającą wierzchołki osiągalne z MDR (w tym momencie zawiera jeden wierzchołek, więc na początku w kolejce będą sąsiedzi początkowego wierzchołka), o priorytecie najmniejszego kosztu dotarcia do danego wierzchołka z MDR.
- Powtarzaj, dopóki drzewo nie obejmuje wszystkich wierzchołków grafu:
 - wśród nieprzetworzonych wierzchołków (spoza obecnego MDR) wybierz ten, dla którego koszt dojścia z obecnego MDR jest najmniejszy.
 - dodaj do obecnego MDR wierzchołek i krawędź realizującą najmniejszy koszt
 - zaktualizuj kolejkę priorytetową, uwzględniając nowe krawędzie wychodzące z dodanego wierzchołka

Wyniki pomiarów Algorytmu Prima

Liczba wierzchołków	Gęstość	Prim - Lista	Prim - macierz
L.p.	%	[μs]	[μs]

	Liczba wierzchołków	Gęstość	Prim - Lista	Prim - macierz
1	20	25%	6	8
2	20	50%	7	10
3	20	75%	9	11
4	20	99%	10	11
5	40	25%	20	29
6	40	50%	23	36
7	40	75%	27	38
8	40	99%	30	35
9	60	25%	38	57
10	60	50%	49	76
11	60	75%	57	76
12	60	99%	65	72
13	80	25%	54	89
14	80	50%	68	121
15	80	75%	83	121
16	80	99%	98	112
17	100	25%	82	132
18	100	50%	104	183
19	100	75%	130	182
20	100	99%	154	168

Opis ogólny Algorytmu Kruskala

Schemat działania:

- Utwórz las L z wierzchołków oryginalnego grafu – każdy wierzchołek jest na początku osobnym drzewem.
- Utwórz zbiór S zawierający wszystkie krawędzie oryginalnego grafu.
- Dopóki S nie jest pusty oraz L nie jest jeszcze drzewem rozpinającym:
 - Wybierz i usuń z S jedną z krawędzi o minimalnej wadze.
 - Jeśli krawędź ta łączyła dwa różne drzewa, to dodaj ją do lasu L, tak aby połączyla dwa odpowiadające drzewa w jedno.
 - W przeciwnym wypadku odrzuć ją.

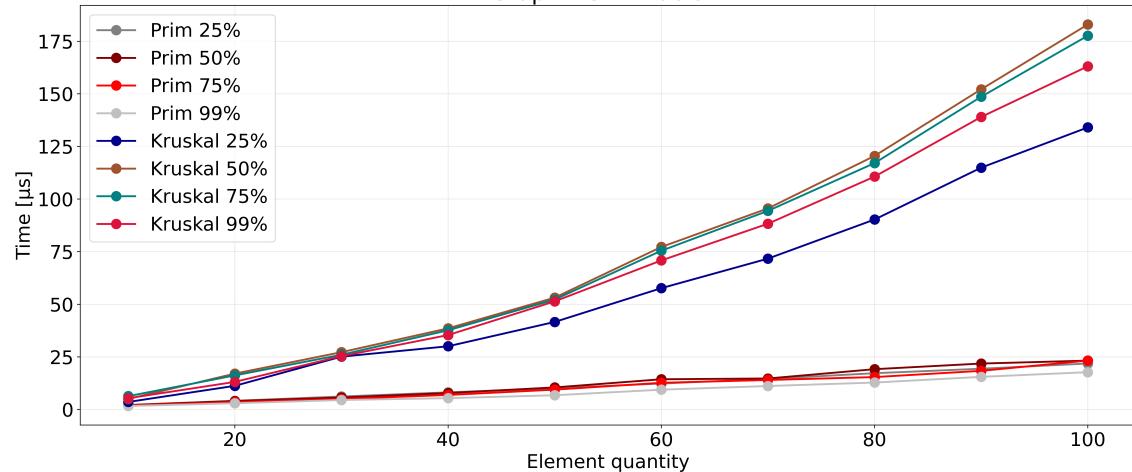
Po zakończeniu algorytmu L jest minimalnym drzewem rozpinającym.

Wyniki pomiaru Algorytmu Kruskala

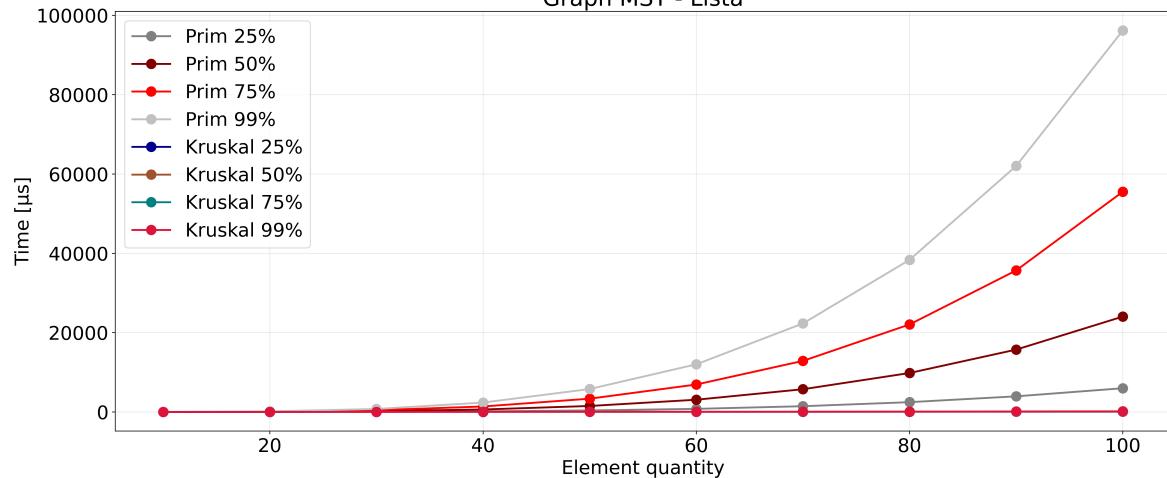
L.p.	Liczba wierzchołków	Gęstość	Kruskal - Lista	Kruskal - macierz
	j	%	[μs]	[μs]
1	20	25%	19	8
2	20	50%	56	20
3	20	75%	113	34
4	20	99%	182	51
5	40	25%	196	60
6	40	50%	714	185
7	40	75%	1538	374
8	40	99%	2629	624
9	60	25%	895	236
10	60	50%	3407	821
11	60	75%	7497	1787
12	60	99%	12935	3058
13	80	25%	2747	687
14	80	50%	10593	2532
15	80	75%	23602	5606
16	80	99%	40736	9668
17	100	25%	6512	1590
18	100	50%	26254	6094
19	100	75%	59099	14018
20	100	99%	110081	25117

Wykresy MST

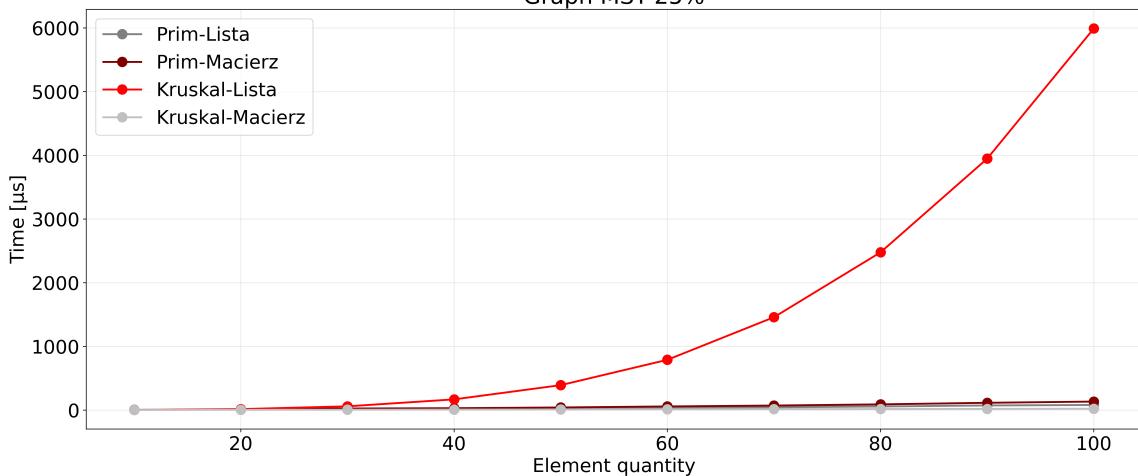
Graph MST - Macierz

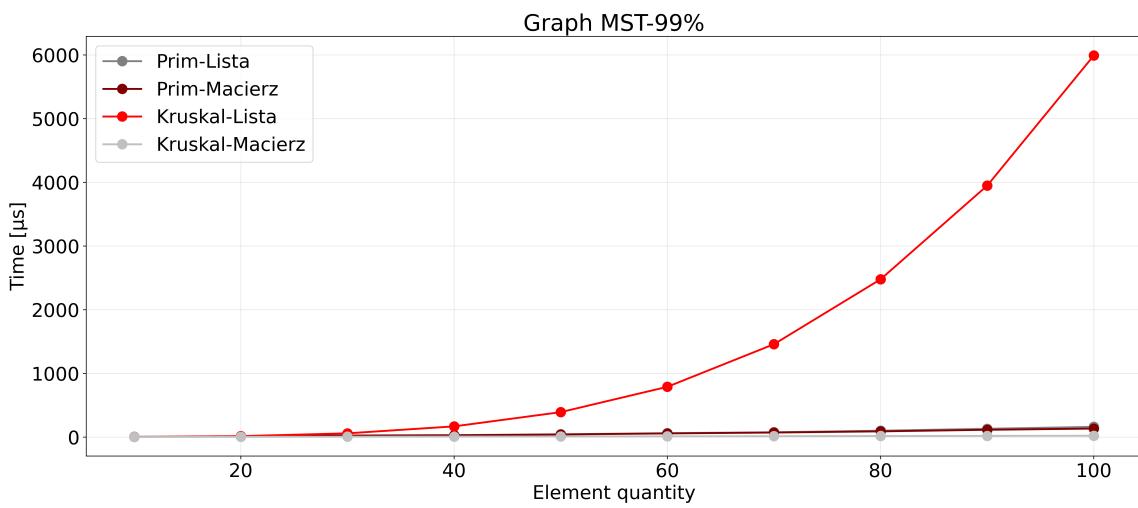
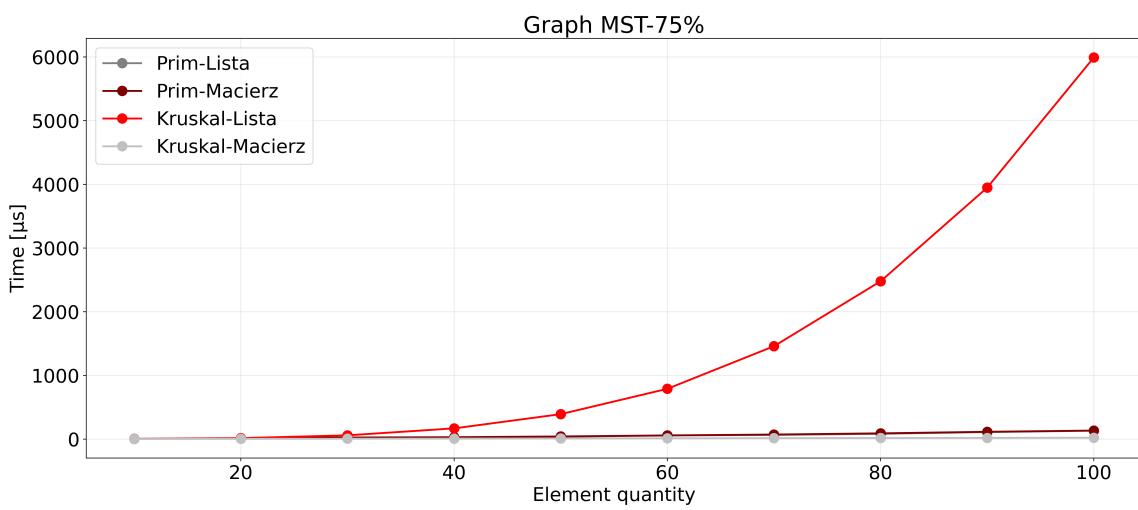
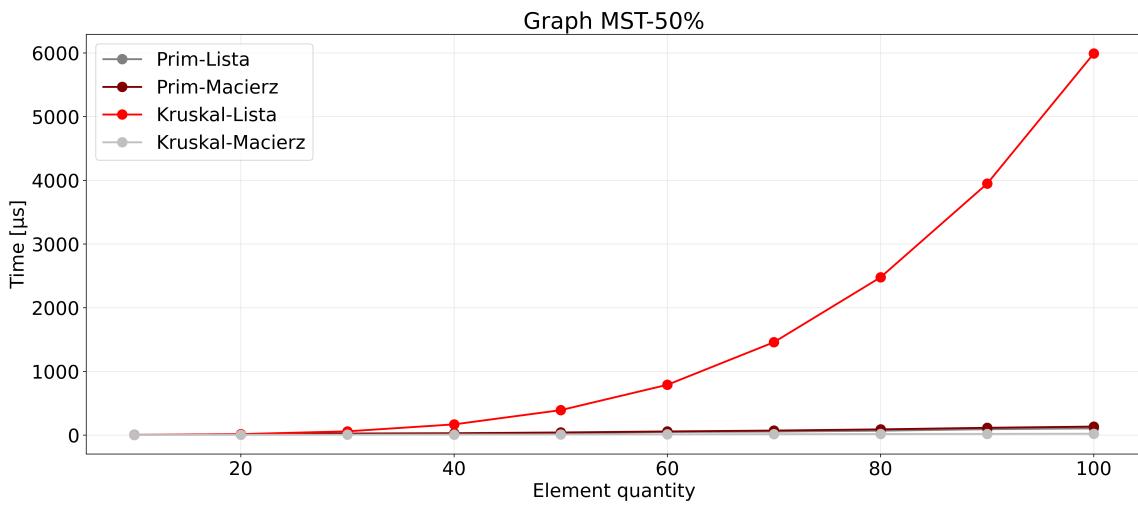


Graph MST - Lista



Graph MST-25%





Wnioski MST

Na obydwu strukturach algorytm Prima okazał się znacznie szybszy, można zauważać że zachodzą delikatne zmiany w czasie (zwiększenie) gdy zwiększa się ilość krawędzi przy implementacji listowej w algorytmie Prima lecz to i tak niewiele zmiany w porównaniu do czasu trwania algorytmu Kruskala.

Wyznaczanie najkrótszej ścieżki w grafie

Opis ogólny Algorytmu Dijkstry

Mając dany graf z wyróżnionym wierzchołkiem (źródłem) algorytm znajduje odległości od źródła do wszystkich pozostałych wierzchołków. Łatwo zmodyfikować go tak, aby szukał wyłącznie (najkrótszej) ścieżki do jednego ustalonego wierzchołka, po prostu przerywając działanie w momencie dojścia do wierzchołka docelowego, bądź transponując tablicę incydencji grafu. Algorytm Dijkstry znajduje w grafie wszystkie najkrótsze ścieżki pomiędzy wybranym wierzchołkiem a wszystkimi pozostałymi, przy okazji wyliczając również koszt przejścia każdej z tych ścieżek.

Algorytm Dijkstry jest przykładem algorytmu zachłannego

Wyniki pomiaru Algorytmu Dijkstry

L.p.	Liczba wierzchołków	Gęstość	Dijkstra - Lista	Dijkstra - macierz
	j	%	[μs]	[μs]
1	20	25%	7	9
2	20	50%	7	11
3	20	75%	8	10
4	20	99%	9	9
5	40	25%	23	34
6	40	50%	28	44
7	40	75%	30	40
8	40	99%	33	33
9	60	25%	44	69
10	60	50%	53	91
11	60	75%	60	81
12	60	99%	69	66
13	80	25%	62	112
14	80	50%	80	151
15	80	75%	95	131
16	80	99%	112	105
17	100	25%	100	176
18	100	50%	130	240
19	100	75%	153	207
20	100	99%	183	165

Algorytm Bellmana-Forda

dea algorytmu opiera się na metodzie relaksacji (dokładniej następuje relaksacja $|V| - 1$ razy każdej z krawędzi).

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). Za tę ogólność płaci się jednak wyższą złożonością czasową. Działa on w czasie $O(|V| \cdot |E|)$ [1].

Algorytm może być wykorzystywany także do sprawdzania, czy w grafie występują ujemne cykle osiągalne ze źródła

Schemat działania:

- Utwórz las L z wierzchołków oryginalnego grafu – każdy wierzchołek jest na początku osobnym drzewem.
- Utwórz zbiór S zawierający wszystkie krawędzie oryginalnego grafu.
- Dopóki S nie jest pusty oraz L nie jest jeszcze drzewem rozpinającym:
 - Wybierz i usuń z S jedną z krawędzi o minimalnej wadze.
 - Jeśli krawędź ta łączyła dwa różne drzewa, to dodaj ją do lasu L , tak aby połączyła dwa odpowiadające drzewa w jedno.
 - W przeciwnym wypadku odrzuć ją.

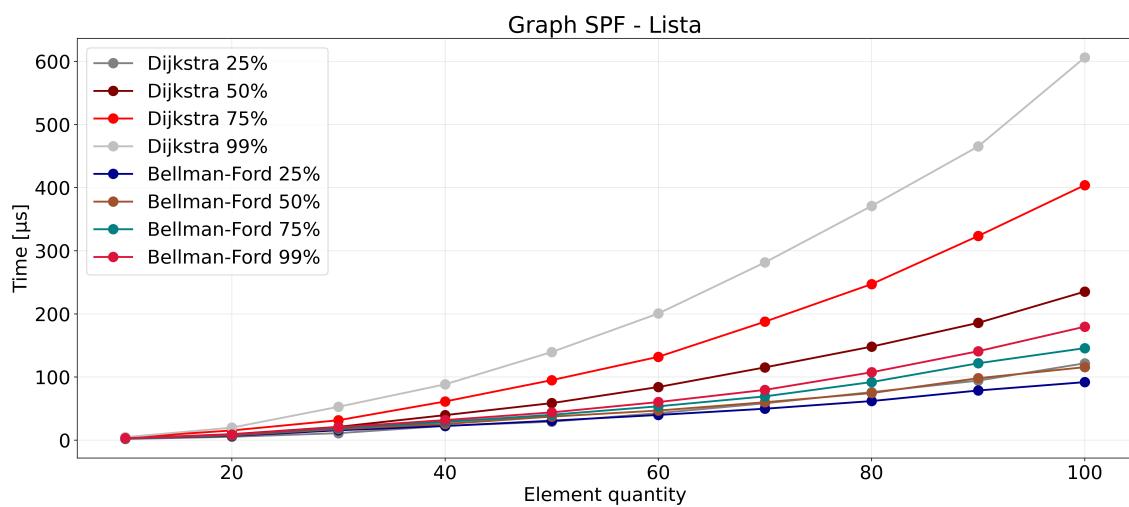
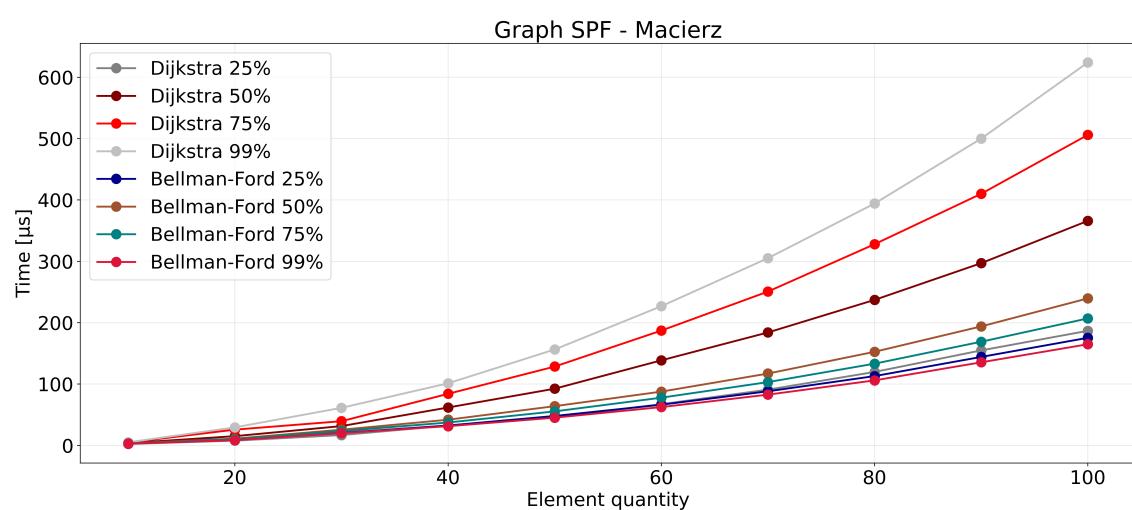
Po zakończeniu algorytmu L jest minimalnym drzewem rozpinającym.

Wyniki pomiaru Algorytmu Bellmana-Forda

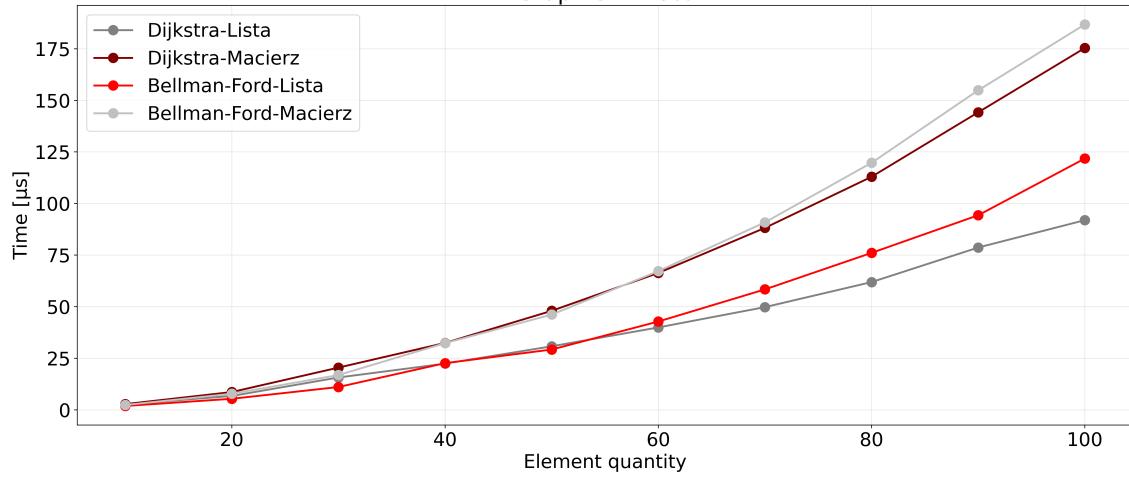
L.p.	Liczba wierzchołków	Gęstość	Bellman-Ford - Lista	Bellman-Ford - macierz
1	j	%	[μs]	[μs]
1	20	25%	9	11
2	20	50%	17	21
3	20	75%	26	28
4	20	99%	35	37
5	40	25%	41	50
6	40	50%	84	99
7	40	75%	124	132
8	40	99%	161	162
9	60	25%	95	116
10	60	50%	187	214
11	60	75%	285	302
12	60	99%	377	375

Liczba wierzchołków	Gęstość	Bellman-Ford - Lista	Bellman-Ford - macierz
13	80	25%	173
14	80	50%	344
15	80	75%	549
16	80	99%	728
17	100	25%	279
18	100	50%	555
19	100	75%	848
20	100	99%	1117
			1123

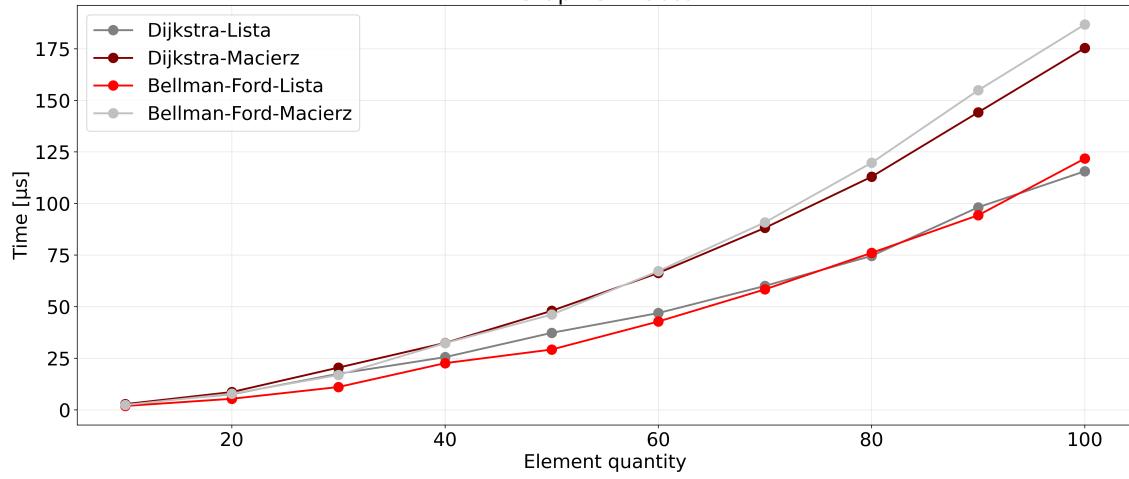
Wykresy SPF



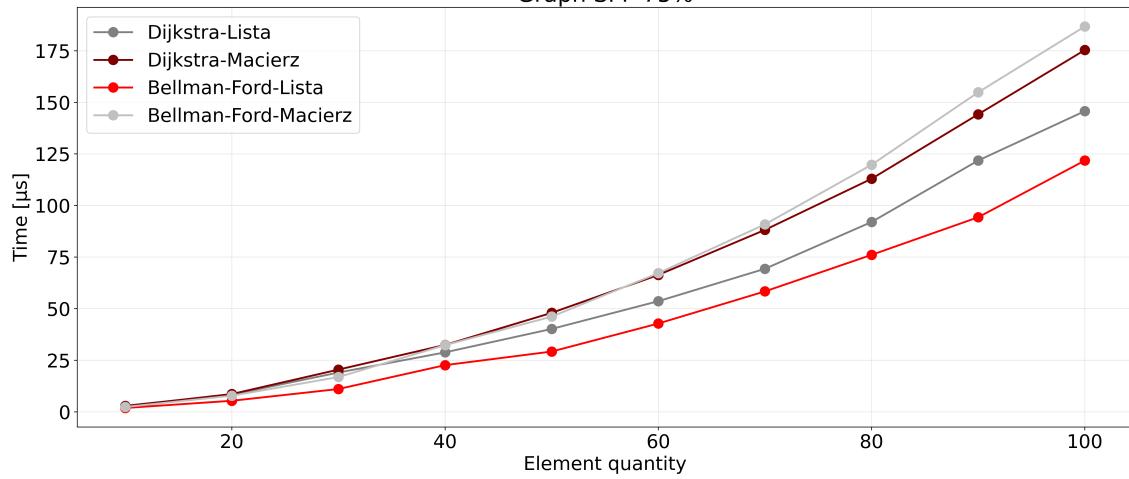
Graph SPF-25%

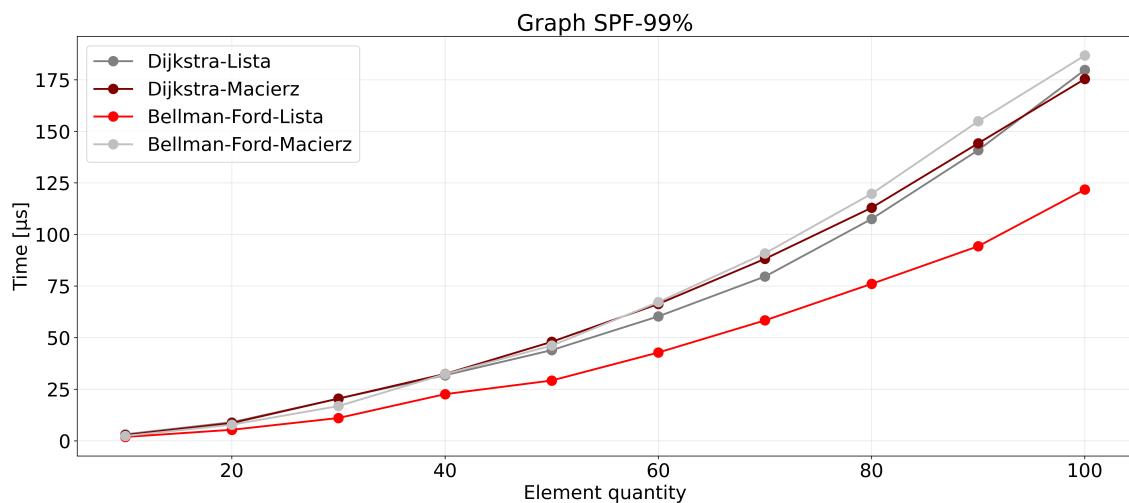


Graph SPF-50%



Graph SPF-75%





Wnioski SPF

Algorytm Dijkstry okazał się lepszy w obydwu przypadkach, w macierzy sąsiedztwa jak i w liście następników. Można zauważyć że wersja algorytmu Dijkstry znaczco spowalnia na strukturze listowej przy zwiększeniu ilości krawędzi.

Wnioski końcowe

Podsumowując wykonane pomiary większość wyszła zgodnie z przewidywaniami, możemy zauważyć że implementacja Macierzy sąsiedztwa jest znacznie bardziej efektywna od implementacji listy następców, więc lepiej przedstawiać grafy w komputerach jako macierze sąsiedztwa.

Bibliografia

[Jarosław Mierzwa](#)

[Wikipedia - Adjacency matrix](#)

[Eduinf Waw - Reprezentacje grafów](#)

[Eduinf Waw - MST](#)

[Wikipedia - Prim](#)

[Wikipedia - Kruskal](#)

[Wikipedia - Dijkstra](#)

[Eduinf Waw - Dijkstra](#)

[Wikipedia - Bellman-Ford](#)

[Eduinf Waw - Bellman-Ford](#)