Termin zajęć: wt. TN godz. 17:05
Prowadzący:

Jakub Balcerzak

Struktury danych i złożoność obliczeniowa

Projekt 2

1. Opis projektu, założenia, pomiary:

Celem projektu była implementacja algorytmów grafowych w tym: algorytmów Prima, Kruskala dla problemu minimalnego drzewa rozpinającego oraz Dijkstry, Bellmana-Forda dla problemu najkrótszej ścieżki z jednym źródłem.

Dla problemu MST graf musi być spójny oraz nieskierowany. W przypadku najkrótszej ścieżki musi być on skierowany i nie powinien zawierać cykli ujemnych.

Tworzony graf reprezentowany jest w pamięci komputera na dwa sposoby, jako macierz sąsiedztwa oraz lista sąsiedztwa.

W przypadku pierwszym założono, że wagi krawędzi reprezentowane są przez wartości całkowite, a liczba 0 zarezerwowana została na umowną wartość wskazującą na brak krawędzi pomiędzy dwoma wierzchołkami.

W programie zaimplementowano funkcję *createList()* , która na podstawie macierzy sąsiedztwa generuje kolejne listy sąsiadów. Zatem, aby graf był reprezentowany przez listę, wpierw konieczne jest istnienie macierzy sąsiedztwa.

Program umożliwia generację grafu (punkt 2.) oraz wczytanie jego struktury z pliku .txt, w którym w pierwszej linii znajdują się rozdzielone spacjami odpowiednio: liczba krawędzi, liczba wierzchołków, wierzchołek początkowy, wierzchołek końcowy. Kolejne linie zawierają informacje o krawędziach w kolejności: wierzchołek początkowy, wierzchołek końcowy, waga krawędzi. Każda z wartości rozdzielona musi być znakiem separatora. Dodatkowo ze względów praktycznych utworzono dwie oddzielne funkcje realizujące wczytywanie grafu nieskierowanego lub skierowanego.

Pomiary przeprowadzono w pętli wykonującej się stukrotnie, zebrane wyniki zostały uśrednione. Przy każdym obiegu generowany był nowy graf, a sam pomiar obejmował jedynie część kodu odpowiedzialną za wykonanie się danego algorytmu. W celu minimalizacji narzutu czasowego pominięto wyświetlanie wyników.

Podczas pomiarów należało dla każdego algorytmu wyznaczyć 5 reprezentatywnych ilości wierzchołków w grafie oraz przeprowadzić pomiary dla jego gęstości odpowiednio: 25%, 50%, 75% i 99%.

Ilości wierzchołków były dobierane eksperymentalnie tak, aby czasy wykonywania się algorytmów były rozsądne. Przy czym dla danego algorytmu w obu reprezentacjach liczby wierzchołków były takie same.

Ponieważ algorytm Kruskala dla listy sąsiedztwa wykonywał się w znaczącym czasie już przy 300 wierzchołkach, a Bellmana-Forda przy 500, przyjęto te liczby, jako górne ograniczenia dla wymienionych algorytmów.

2. Generowanie grafu:

2.1. nieskierowanego:

Niech V – liczba wierzchołków, E – liczba krawędzi, d – gęstość grafu w %, wówczas:

$$E = (d/100) * ((V) * (V-1) / 2)$$

Graf nieskierowany jest generowany dla problemu MST. Należy zadbać o ty, by wystąpiło w nim jakiekolwiek minimalne drzewo rozpinające. Dlatego przyjęto następującą strategię: wpierw tworzona jest gwiazda o środku w wierzchołku zerowym, a jeżeli pula krawędzi się nie wyczerpie, wtem pozostałe krawędzie zostają losowo rozdysponowane pomiędzy pozostałe wierzchołki.

2.2. skierowanego:

Niech V – liczba wierzchołków, E – liczba krawędzi, d – gęstość grafu w %, wówczas:

$$E = (d/100) * ((V) * (V))$$

Dla problemu najkrótszej ścieżki generowany jest graf skierowany, w którym należy zapewnić wystąpienie jakiejkolwiek ścieżki. Wobec czego przyjęto następującą strategię: każdy wierzchołek łączony jest z następnym w kolejności wierzchołkiem, aż ostatni wierzchołek nie zostanie połączony z wierzchołkiem zerowym. Przykład dla V = 5 i d = 20%: 0 –> 1 –> 2 –> 3 –> 4 –> 0. Jeżeli pula krawędzi się nie wyczerpie, wówczas pozostałe krawędzie zostają losowo rozdysponowane pomiędzy pozostałe wierzchołki.

W grafie skierowanym dopuszczalne są pętle.

2.3. wagi krawędzi:

Oprócz losowego wyboru wierzchołków, również losowane zostają wagi krawędzi je łączące. Wagi te losowane są z przedziału [1, 100000] o rozkładzie równomiernym.

3. Minimalne drzewa rozpinające – MST:

3.1. Wstęp teoretyczny:

Szybkość działania algorytmu Prima dla reprezentacji listowej zależy od sposobu implementacji kolejki priorytetowej. Jeżeli kolejka zaimplementowana została, jako kopiec binarny oraz budowana jest z nieuporządkowanej tablicy wówczas inicjalizacja kolejki wynosi O(V). Każda operacja usunięcia korzenia kopca to O(lgV), ponieważ kolejkę opróżniamy do końca, to całkowity czas tej operacji wynosi O(V lgV). Dodatkowo operacja modyfikacji klucza może zostać zaimplementowana ze złożonością O(lgV). Całkowity czas działania algorytmu Prima, przy tak określonych warunkach implementacyjnych wynosi O(E lgV). Można poprawić ten czas do O(E + V lgV), jeżeli zamiast kopca binarnego zostanie użyty kopiec Fibonacciego. Literatura nie podaje złożoności dla reprezentacji macierzowej.

Algorytm Kruskala z kolei zależny jest od sposobu implementacji struktury zbiorów rozłącznych. Czas sortowania krawędzi wynosi O(E lgE), a całkowita złożoność tego algorytmu określana jest jako O(E lgV), a zatem równa jest złożoności algorytmu Prima. Literatura nie precyzuje, jakiej reprezentacji grafu dot. podane złożoności.

3.2. Pomiary:

Wszystkie zmierzone czasy są wyrażane w ms.

Algorytm Prima – macierz sąsiedztwa:

V \ d	25%	50%	75%	99%
500	1,91	2,51	2,13	1,51
875	5,63	7,55	6,29	4,63
1250	11,53	15,33	13,03	9,65
1625	19,44	26,39	22,07	16,42
2000	29,65	39,86	33,42	24,87

Algorytm Prima - lista sąsiedztwa:

V \ d	25%	50%	75%	99%
500	4,74	8,21	11,66	14,48
875	19,33	35,88	51,26	68,49
1250	50,90	97,72	144,56	188,46
1625	110,58	215,37	313,28	414,18
2000	205,44	397,52	587,44	786,93

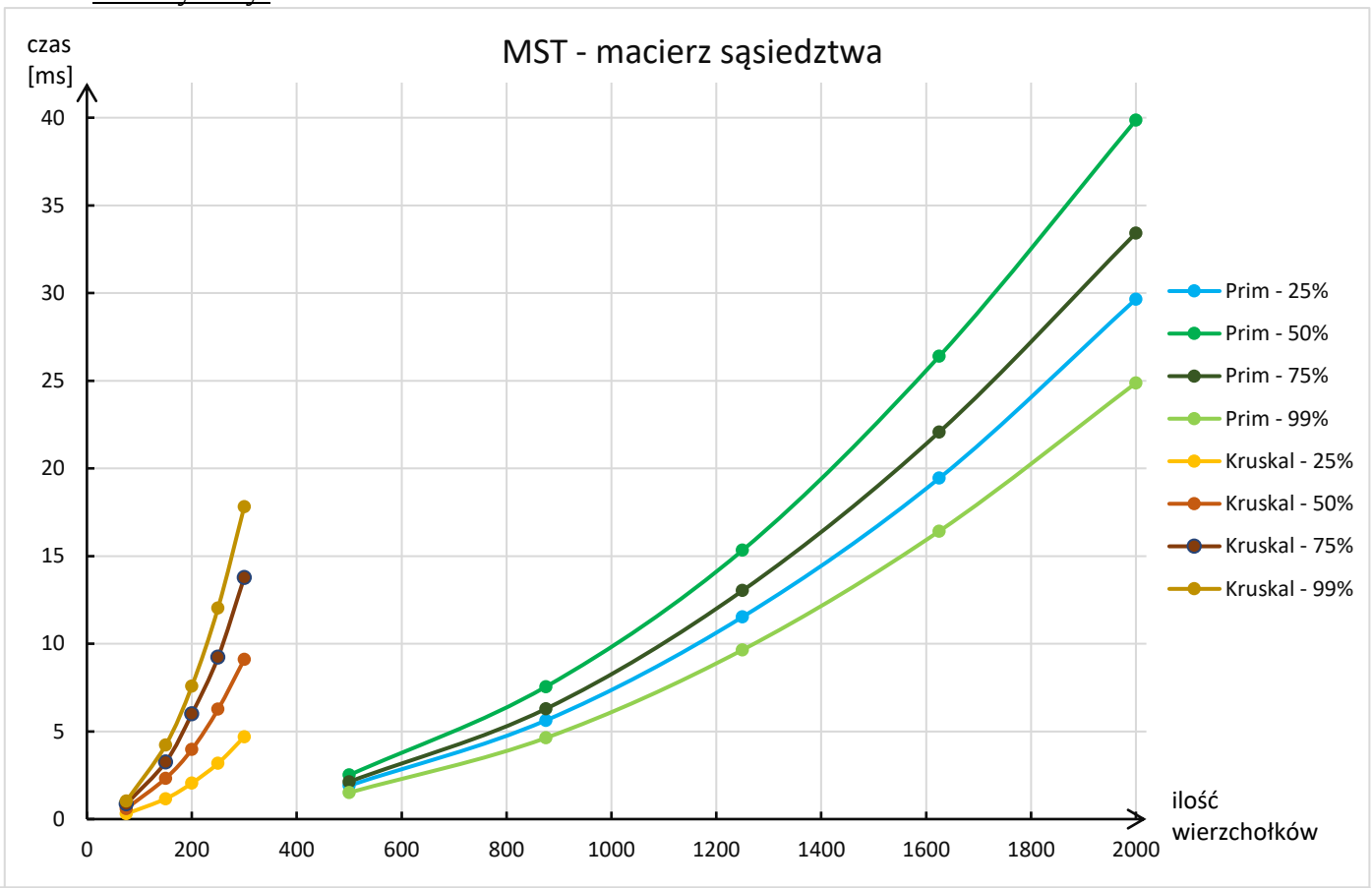
Algorytm Kruskala – macierz sąsiedztwa:

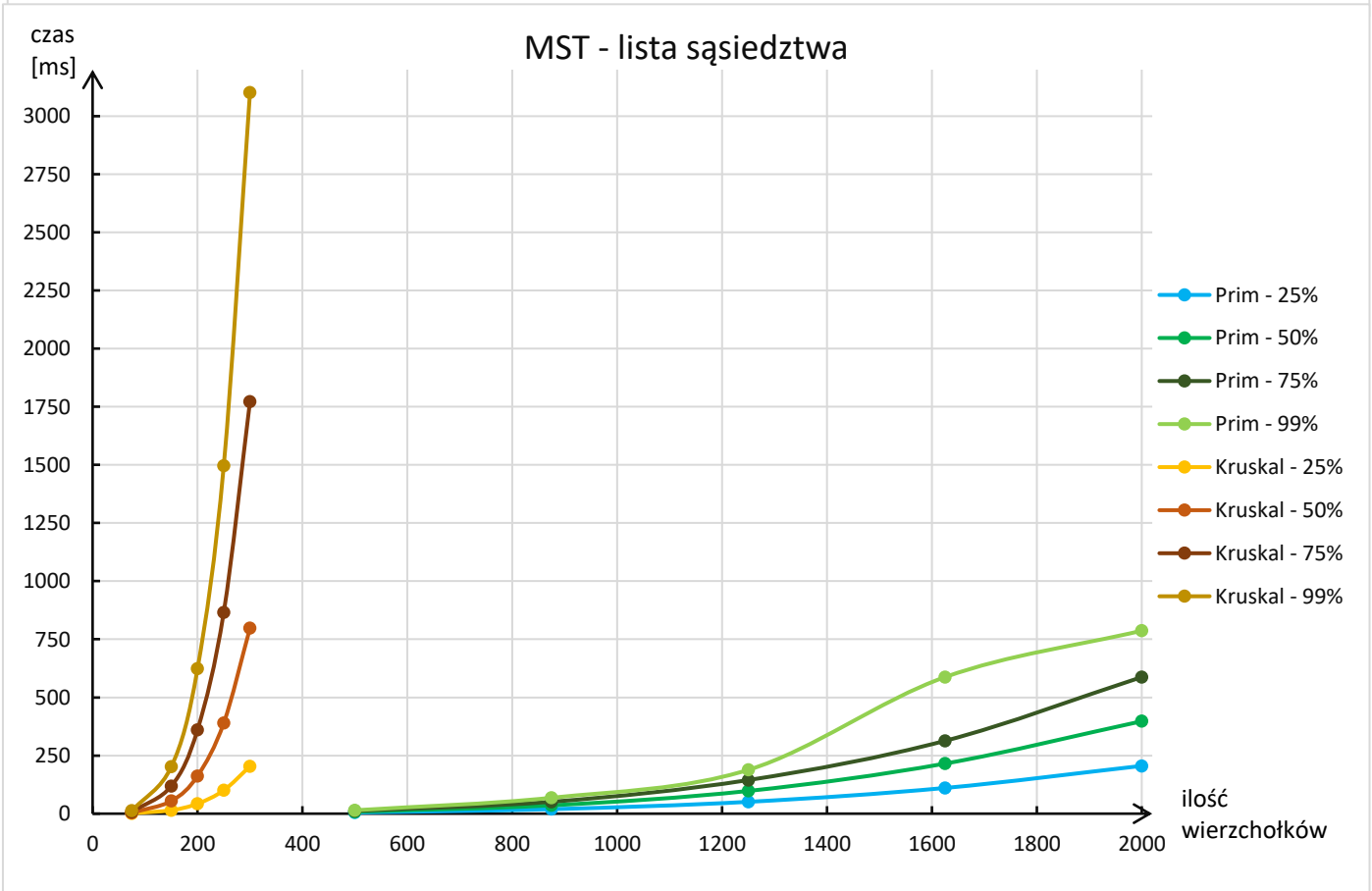
V \ d	25%	50%	75%	99%
75	0,30	0,60	0,86	1,02
150	1,16	2,32	3,25	4,22
200	2,05	3,97	6,00	7,58
250	3,18	6,26	9,23	12,04
300	4,69	9,11	13,78	17,82

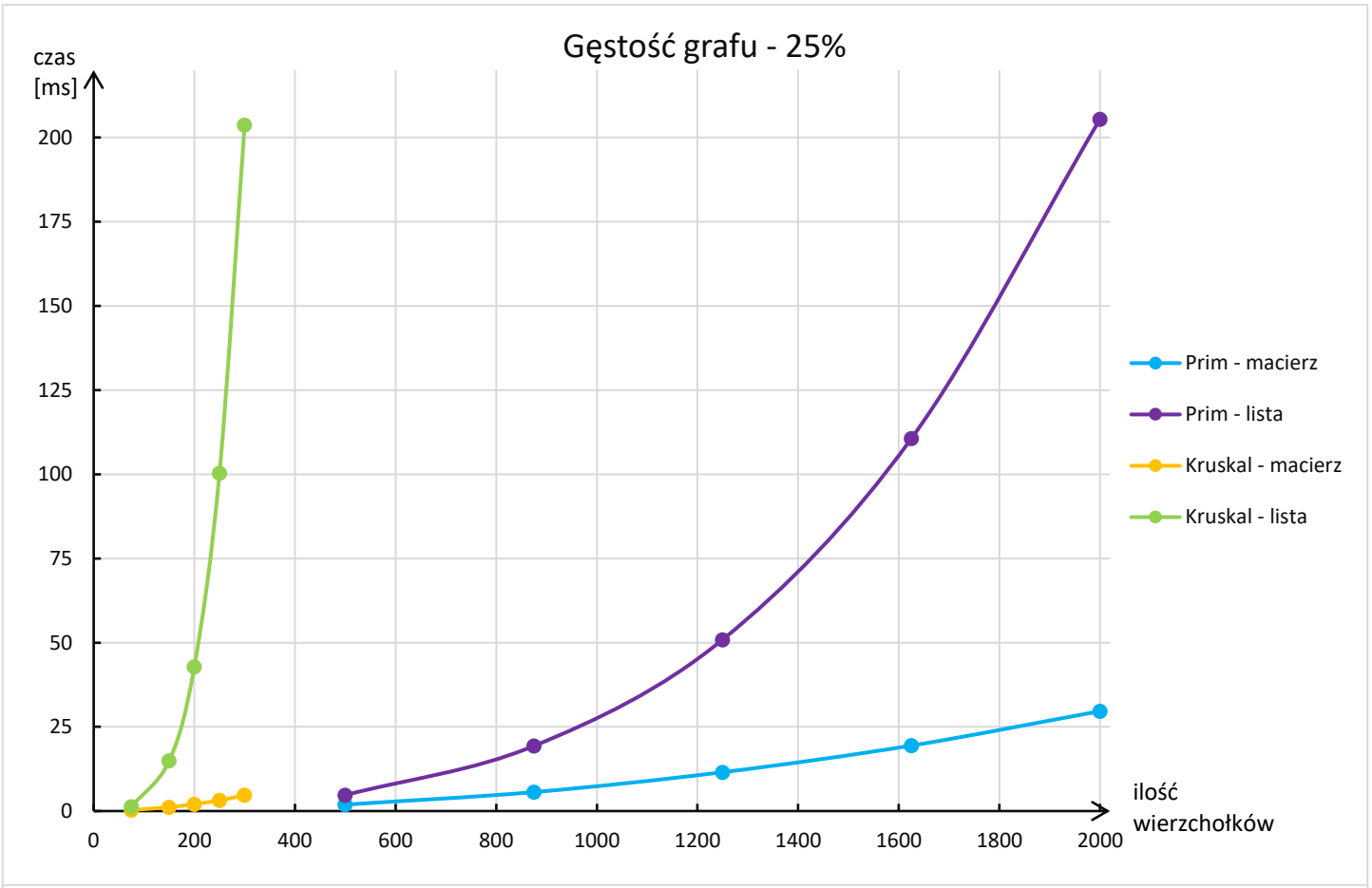
Algorytm Kruskala – lista sąsiedztwa:

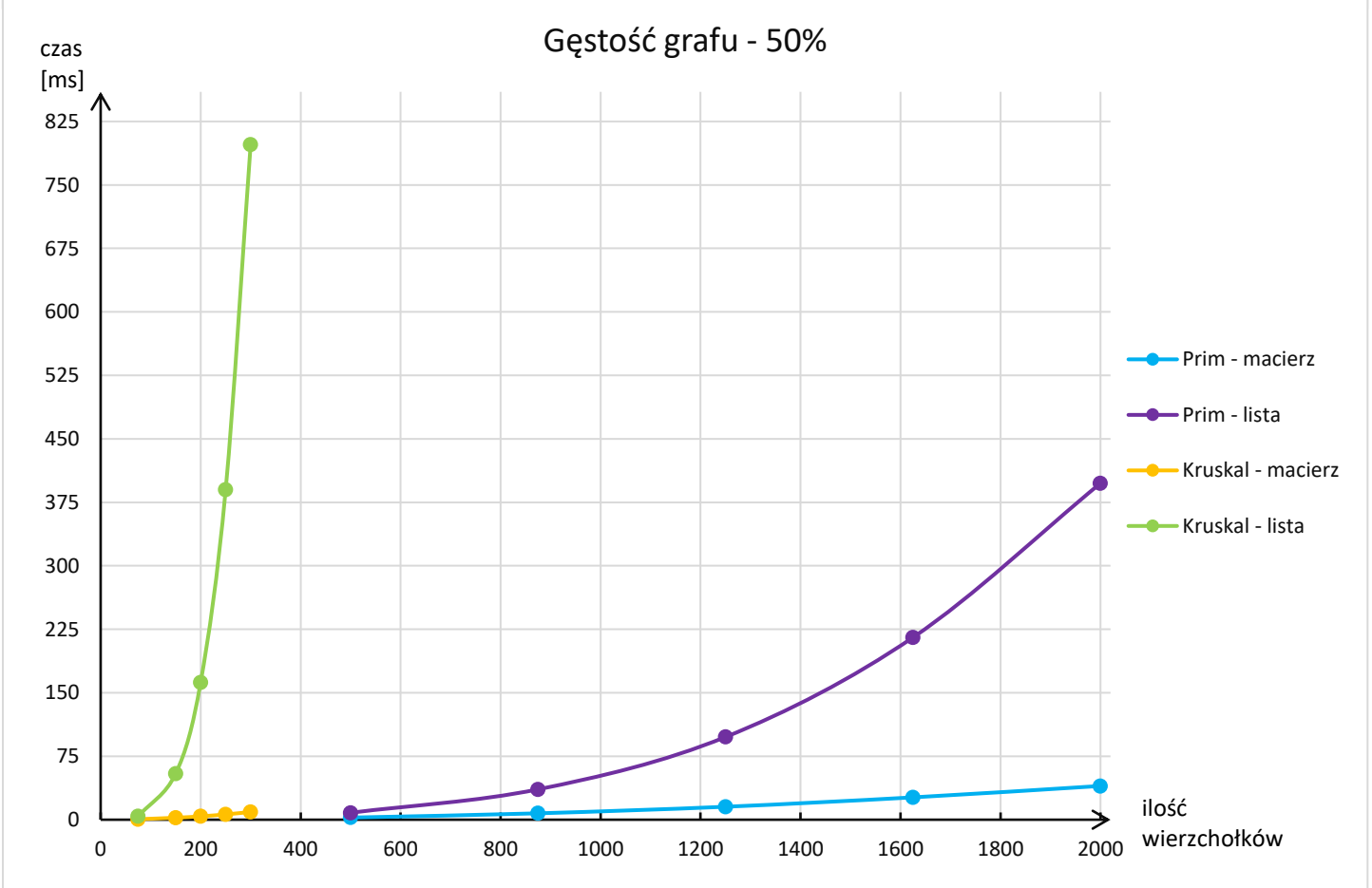
V \ d	25%	50%	75%	99%
75	1,30	3,97	8,09	12,65
150	14,90	54,39	118,39	201,59
200	42,86	162,16	360,54	623,36
250	100,32	389,79	865,69	1495,92
300	203,69	797,63	1772,50	3101,22

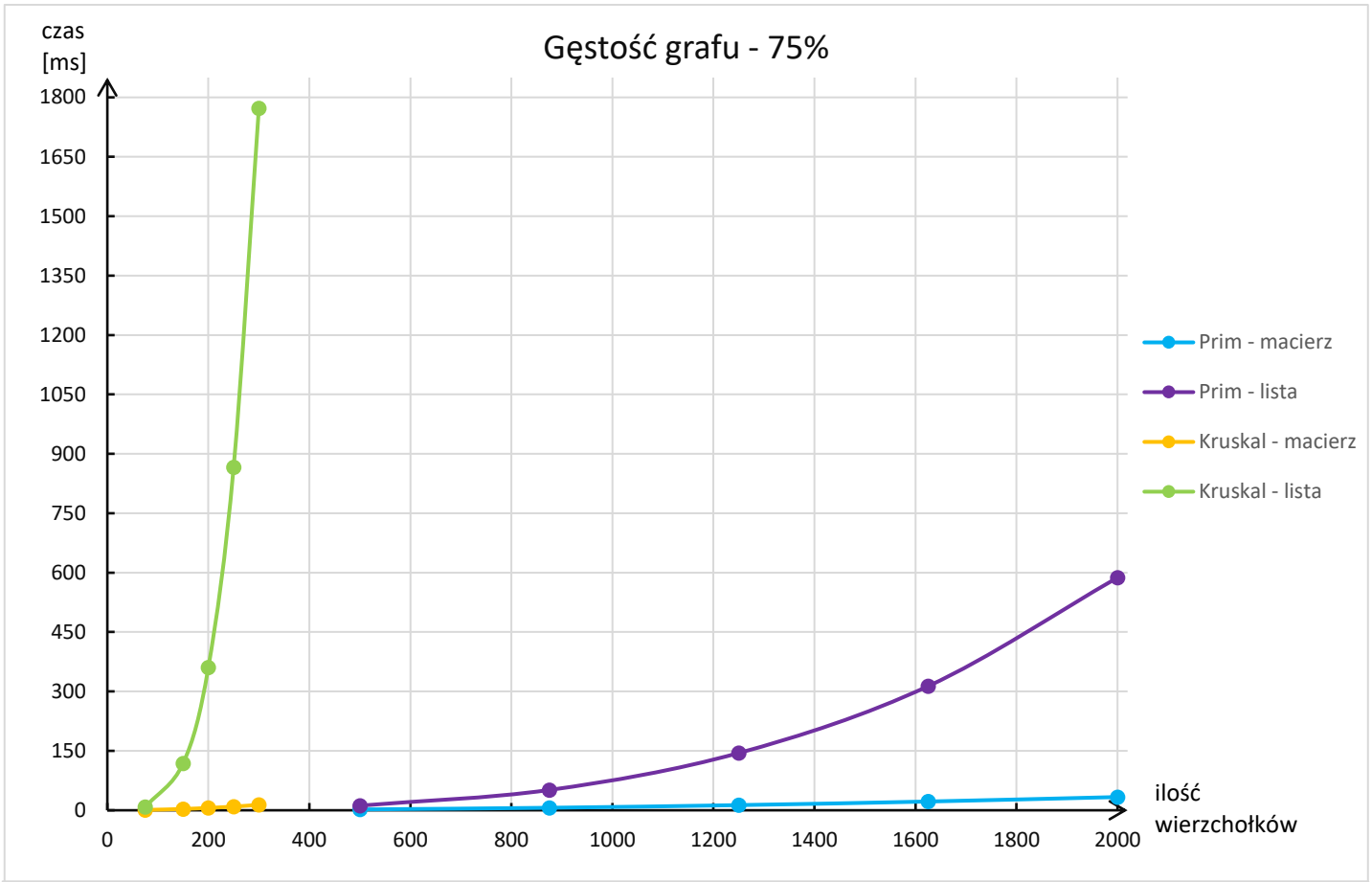
3.3. Wykresy:

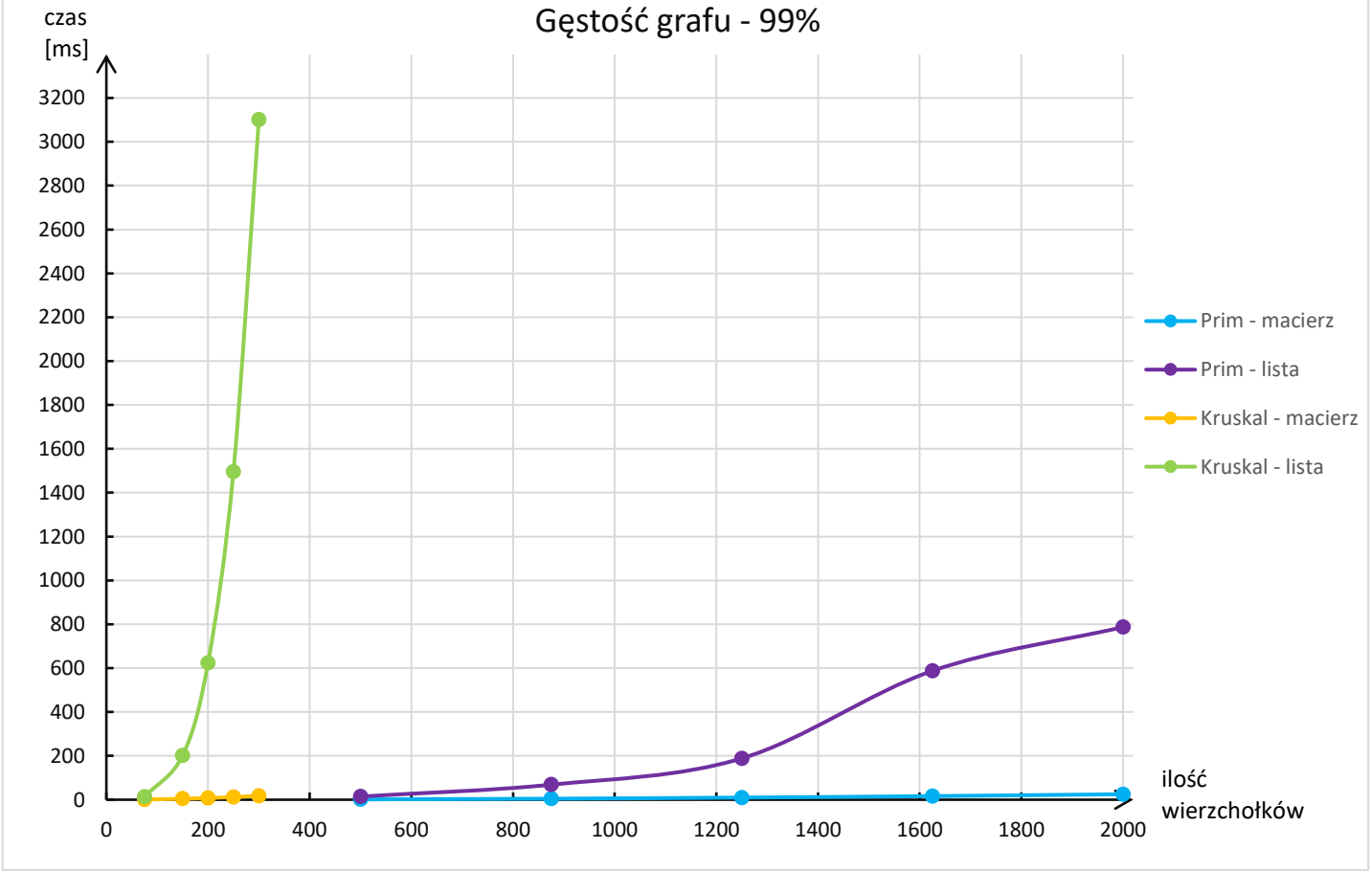

MST - macierz sąsiedztwa
czas [ms]
0
5
10
15
20
25
30
35
40
0
200
400
600
800
1000
1200
1400
1600
1800
2000
ilość wierzchołków
Prim - 25%
Prim - 50%
Prim - 75%
Prim - 99%
Kruskal - 25%
Kruskal - 50%
Kruskal - 75%
Kruskal - 99%


MST - lista sąsiedztwa
czas [ms]
0
250
500
750
1000
1250
1500
1750
2000
2250
2500
2750
3000
0
200
400
600
800
1000
1200
1400
1600
1800
2000
ilość wierzchołków
Prim - 25%
Prim - 50%
Prim - 75%
Prim - 99%
Kruskal - 25%
Kruskal - 50%
Kruskal - 75%
Kruskal - 99%


Gęstość grafu - 25%
czas [ms]
200
175
150
125
100
75
50
25
0
0
200
400
600
800
1000
1200
1400
1600
1800
2000
ilość wierzchołków
Prim - macierz
Prim - lista
Kruskal - macierz
Kruskal - lista


Gęstość grafu - 50%
czas [ms]
825
750
675
600
525
450
375
300
225
150
75
0
0
200
400
600
800
1000
1200
1400
1600
1800
2000
ilość wierzchołków
Prim - macierz
Prim - lista
Kruskal - macierz
Kruskal - lista


Gęstość grafu - 75%
czas [ms]
1800
1650
1500
1350
1200
1050
900
750
600
450
300
150
0
0
200
400
600
800
1000
1200
1400
1600
1800
2000
ilość wierzchołków
Prim - macierz
Prim - lista
Kruskal - macierz
Kruskal - lista


Gęstość grafu - 99%
czas [ms]
3200
3000
2800
2600
2400
2200
2000
1800
1600
1400
1200
1000
800
600
400
200
0
0
200
400
600
800
1000
1200
1400
1600
1800
2000
ilość wierzchołków
Prim - macierz
Prim - lista
Kruskal - macierz
Kruskal - lista

3.4. Wnioski:

Dla algorytmu Prima zaimplementowano kolejkę priorytetową w oparciu o kopiec binarny, jego budowa odbywa się poprzez wstawianie nowych węzłów, zatem złożoność inicjalizacji kolejki wynosi O(V lgV). Proces aktualizacji kluczy znajdujących się w kolejce posiada pesymistyczny czas O(V), gdyż przed wykonaniem modyfikacji, przeszukana zostaje tablica w poszukiwaniu odpowiedniego węzła.

Kształt wykresów w przypadku algorytmu Prima dla obu reprezentacji grafów jest podobny do kształtu funkcji f(x) = x lgx. Można wnioskować zatem, że złożoność zaimplementowanego algorytmu w przybliżeniu odpowiada złożoności teoretycznej O(E lgV).

O ile w przypadku listy sąsiedztwa dla rosnącej gęstości grafu wzrasta czas wykonywania się algorytmu, o tyle zaskakuje fakt, że w przypadku reprezentacji macierzowej, najdłuższe czasy algorytm Prima osiąga dla gęstości grafu równej 50%, a najmniejsze dla gęstości 99%. Ponieważ sposób pomiaru czasu był identyczny dla każdego rozpatrywanego przypadku, należy stwierdzić, że wspomniane zjawisko musi być skutkiem sposobu implementacji algorytmu, a nie błędem w pomiarach.

Dodatkowo należy zauważyć, że dla wykresów odpowiadających badanym gęstością zauważalny jest znacznie dłuższy czas wykonywania się algorytmu Prima w przypadku reprezentacji listowej, niż macierzowej. Wytłumaczeniem tego jest fakt, że macierz jest tablicą dwuwymiarową i dostęp do jej elementów po indeksie wynosi O(1) (zajmuje stały czas), a dostęp do elementu na liście sąsiedztwa, wymaga przeglądnięcia i sprawdzenia wszystkich jej elementów.

W przypadku reprezentacji macierzowej również zamiast kolejki priorytetowej, zastosowano tablicę przechowującą klucze, w której indeks tablicy odpowiadał indeksowi wierzchołka. Takie usprawnienie algorytmu jest wymienione również w literaturze oraz innych źródłach.

Do sortowania krawędzi w algorytmie Kruskala wykorzystano istniejącą już kolejkę priorytetową. Proces wstawiania krawędzi wynosić powinien O(E lgE), a usunięcie korzenia kopca powinno zajmować czas O(lgE).

Pomimo tego zauważalne na pierwszy rzut oka są drastyczne różnice w czasie wykonywania się algorytmu Kruskala dla reprezentacji listowej, a macierzowej. Fakt ten wynika ze sposobu implementacji algorytmu.

W przypadku macierzy sąsiedztwa wykorzystany został fakt, że jest on symetryczna względem swojej przekątnej. Zatem podczas wstawiania krawędzi do kolejki nie trzeba było sprawdzać, czy dana krawędź w kolejce już się znalazła. Zapewniało nam to właśnie przetwarzanie połowy macierzy.

Jednak w przypadku listy należało już kontrolować, aby krawędź nie została ponownie dodana. W tym celu zaimplementowano dodatkową funkcję kolejki, która wyszukiwała w tablicy (w kopcu), czy dana krawędź już się w niej znajduje. Czas każdej takiej operacji wynosił O(E), a ponieważ do kolejki wkładane były wszystkie krawędzie złożoność czasowa degenerowała się do $O(E^2)$. Ta degeneracja miała swoje skutki dla całego algorytmu Kruskala dla reprezentacji listowej i sprawiła, że już w przypadku 300 wierzchołków algorytm Kruskala wykonywał się dłużej, niż algorytm Prima dla 2000 wierzchołków.

Co więcej, jak wynika z przeprowadzonych pomiarów algorytm Prima działa szybciej, niż algorytm Kruskala. Zatem podczas implementacji nie udało uzyskać się tożsamych złożoności, o których czytamy w literaturze.

4. Najkrótsza ścieżka z jednym źródłem:

4.1. Wstęp teoretyczny:

Tak jak w przypadku alg. Prima, złożoność czasowa algorytmu Dijkstry również zależy od sposobu implementacji kolejki priorytetowej. Stosując do jej implementacji kopiec binarny oraz wiążąc wierzchołek z elementem kopca poprzez odpowiedni odnośnik – uchwyt, możemy osiągnąć czas całkowity równy $O((V + E) \lg V) = O(E \lg V)$. Czas ten może w najgorszym wypadku wynieść również $O(V^2)$. Z kolei użycie kopców Fibonacciego w miejsce kopca binarnego może skutkować poprawieniem się złożoności czasowej do $O(V \lg V + E)$. Omawiane złożoności dotyczą grafów reprezentowanych, jako listy sąsiedztwa. Literatura nie podaje zmian występujących w przypadku reprezentacji macierzowej.

Dla algorytmu Bellmana – Forda niezależnie od sposobu reprezentacji grafu wszystkie krawędzie przetwarzane są $|V - 1|$ razy oraz dodatkowy raz na koniec, by sprawdzić wystąpienie ujemnych cyklów. Złożoność tego algorytmu wynosi $O(V E)$. Literatura nie podaje sposobów na usprawnienie działania tego algorytmu.

4.2. Pomiary:

Algorytm Dijkstry – macierz sąsiedztwa:

V \ d	25%	50%	75%	99%
500	2,26	2,47	2,16	1,60
750	4,80	5,38	4,85	3,75
1000	8,48	9,40	9,15	6,60
1250	13,32	15,19	13,32	10,34
1500	19,22	21,20	19,54	14,93

Algorytm Dijkstry – lista sąsiedztwa:

V \ d	25%	50%	75%	99%
500	1,23	1,77	2,32	2,70
750	2,67	4,00	5,00	6,04
1000	4,79	6,83	9,18	10,53
1250	7,18	10,84	13,79	16,60
1500	10,72	15,76	20,58	24,54

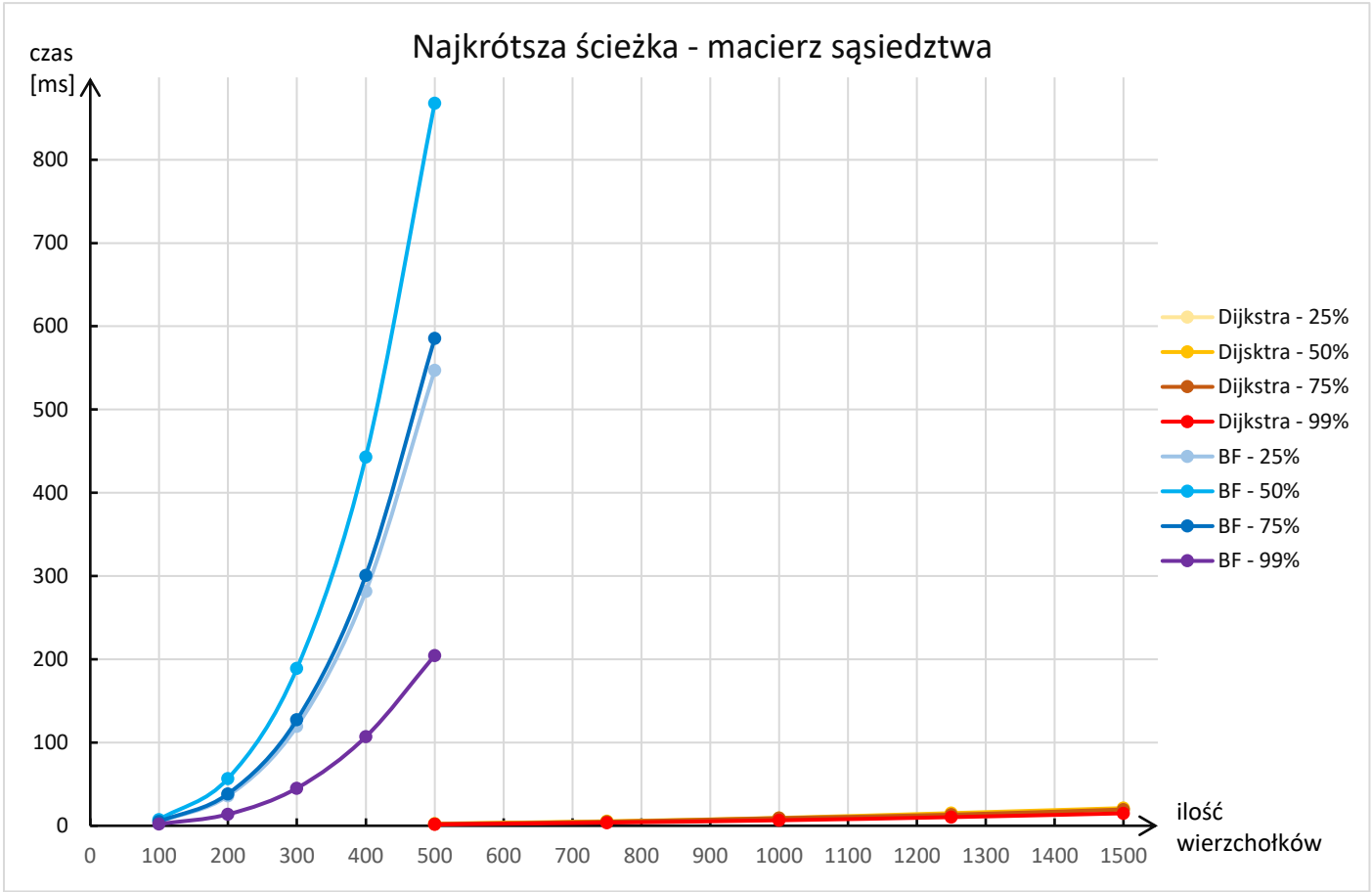
Algorytm Bellmana – Forda – macierz sąsiedztwa:

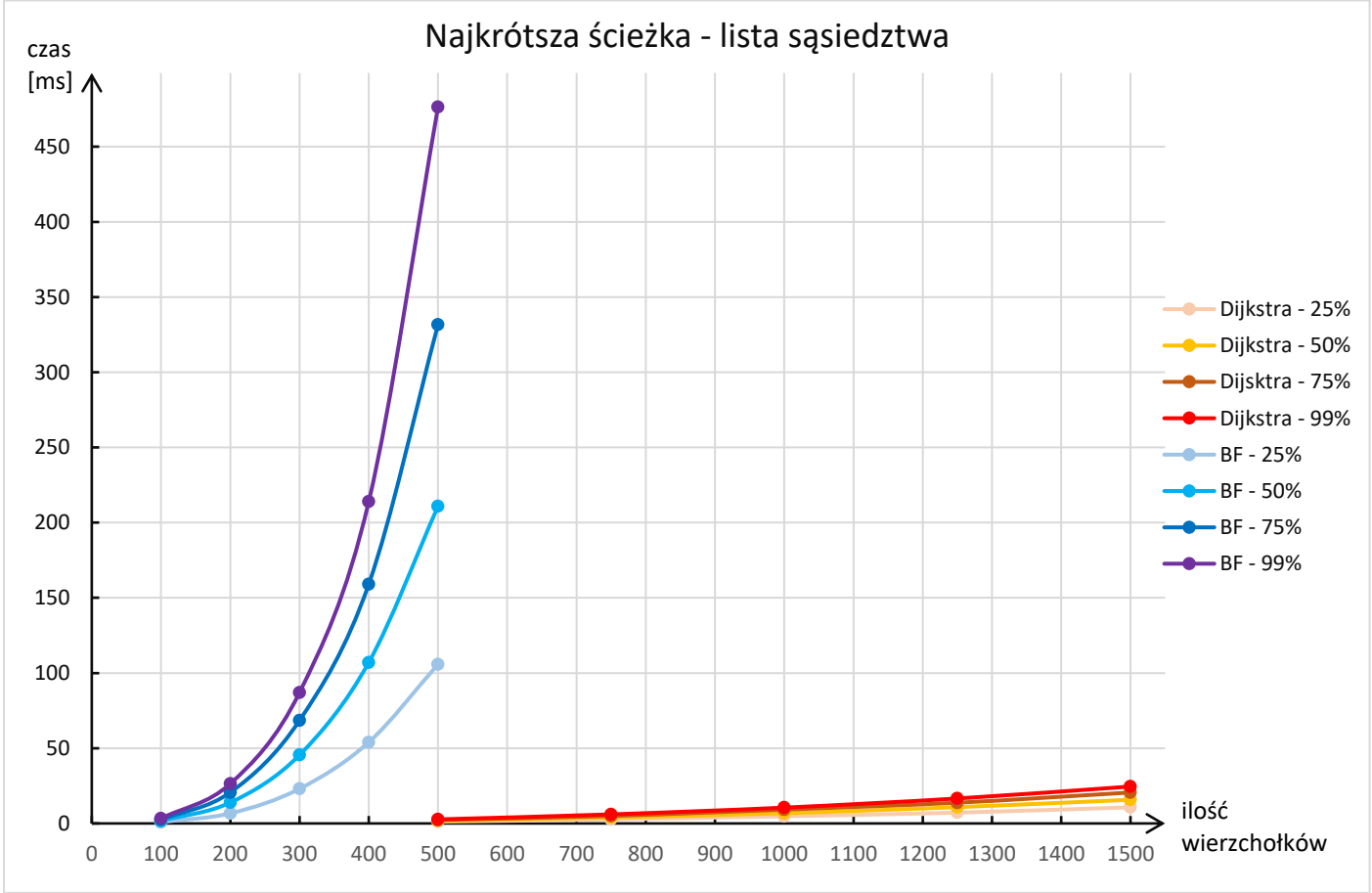
V \ d	25%	50%	75%	99%
100	4,82	7,58	5,01	2,05
200	36,00	56,85	38,41	13,81
300	119,47	188,97	127,36	45,18
400	281,47	443,14	300,86	106,93
500	547,13	868,13	585,57	204,62

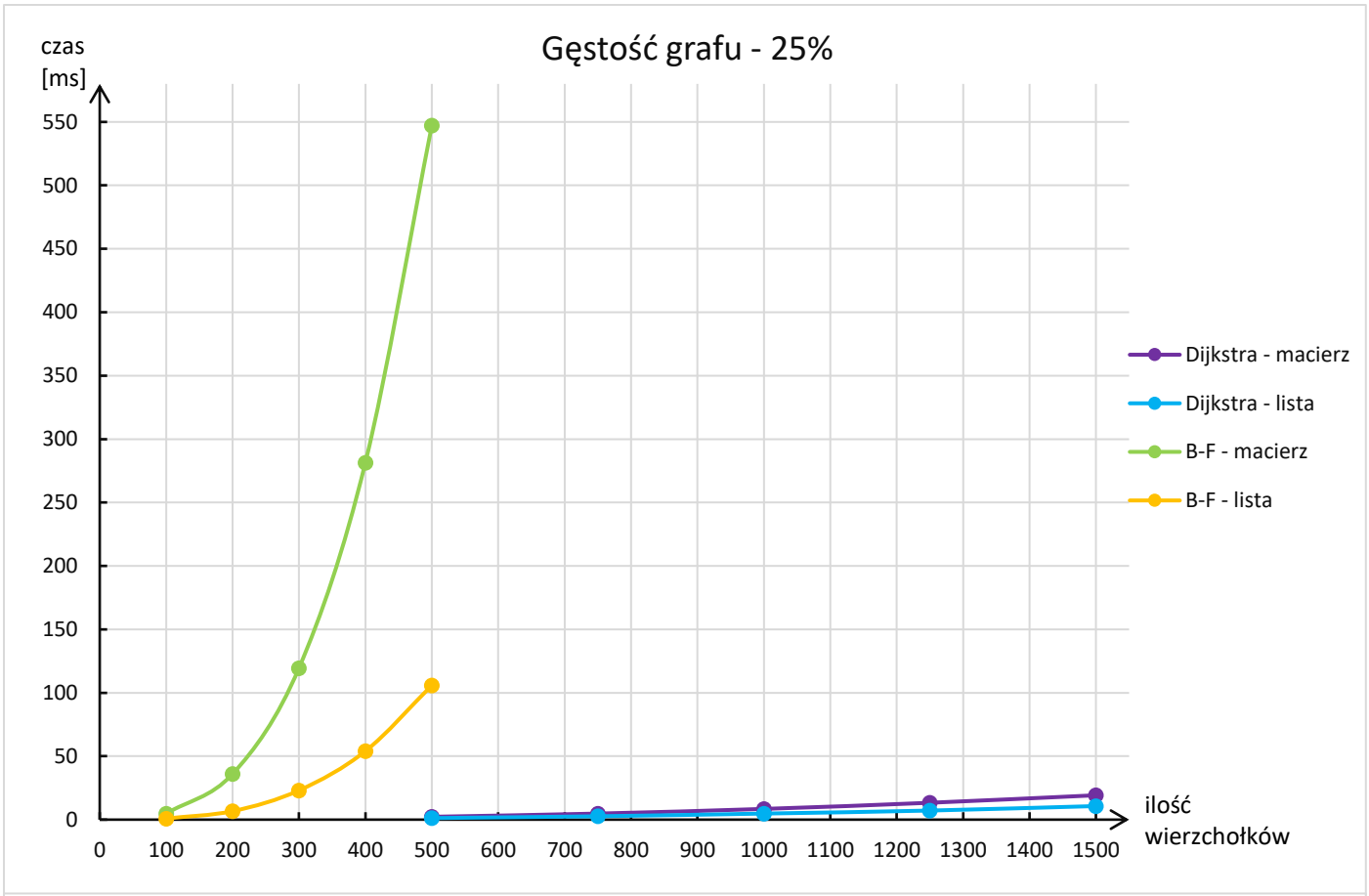
Algorytm Bellmana – Forda – lista sąsiedztwa:

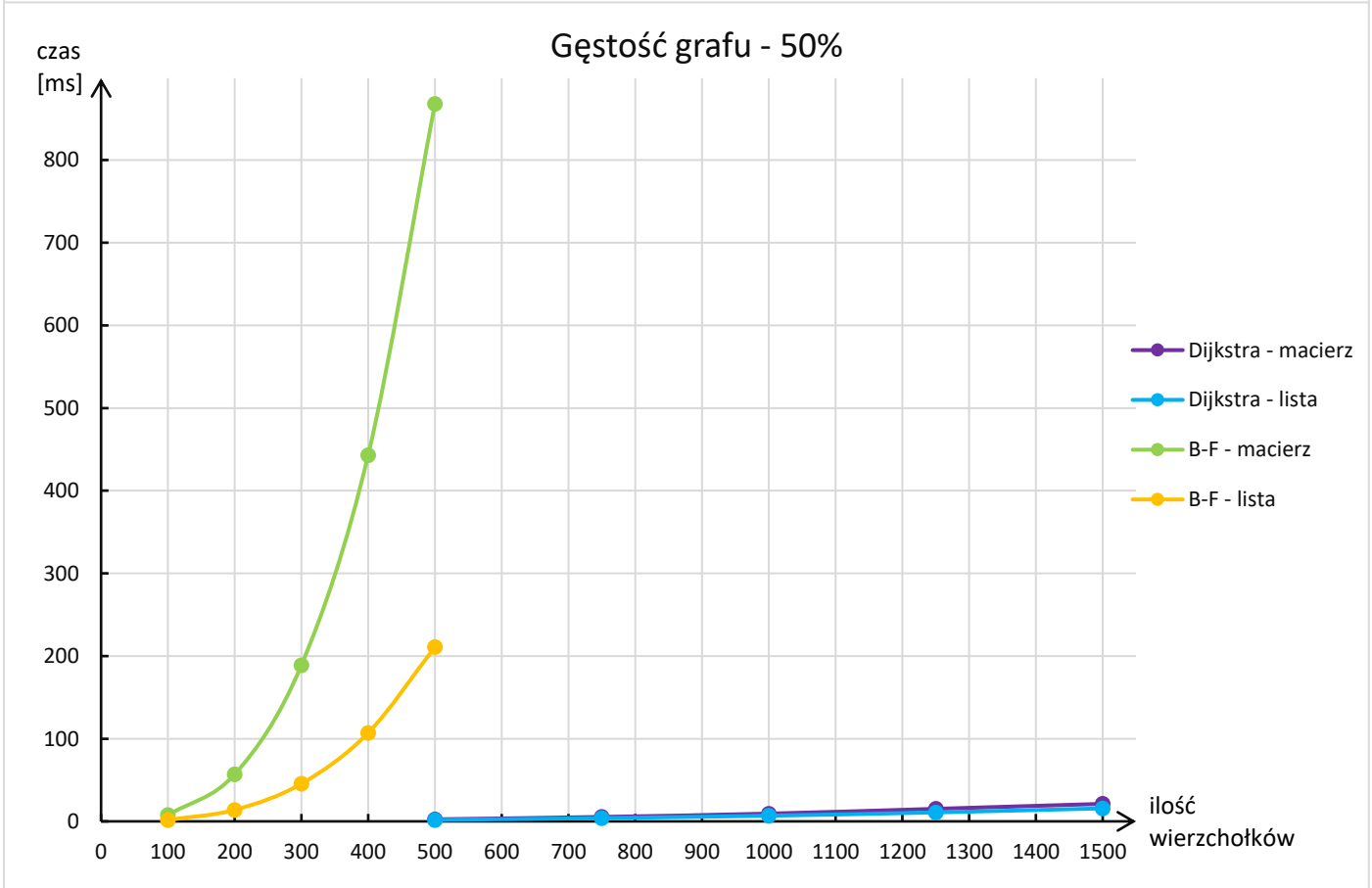
V \ d	25%	50%	75%	99%
100	0,89	1,73	2,50	3,33
200	6,70	13,88	20,65	26,41
300	23,06	45,51	68,45	87,07
400	54,01	107,04	159,08	213,97
500	105,83	210,90	331,67	476,34

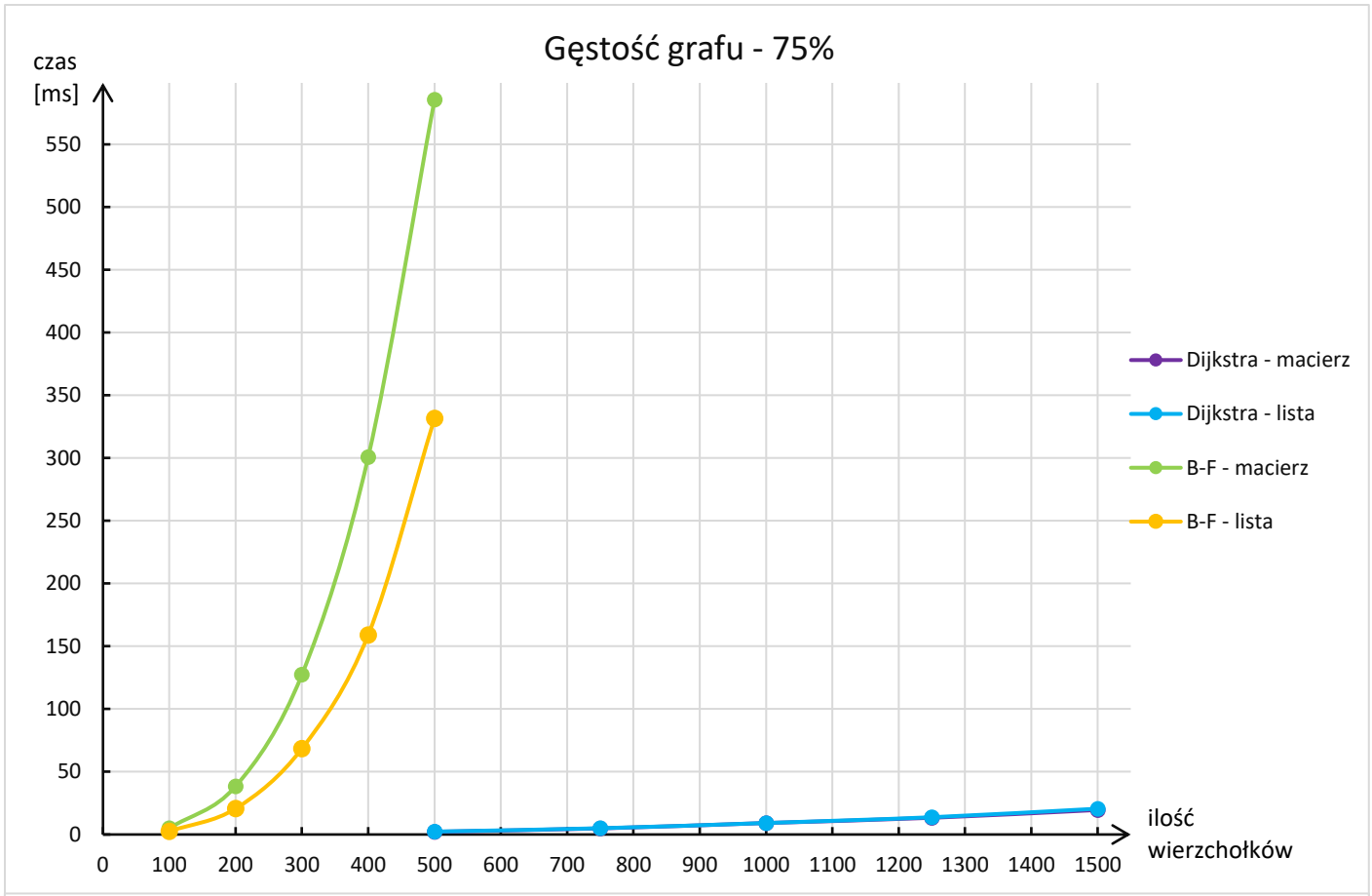
4.3. Wykresy:

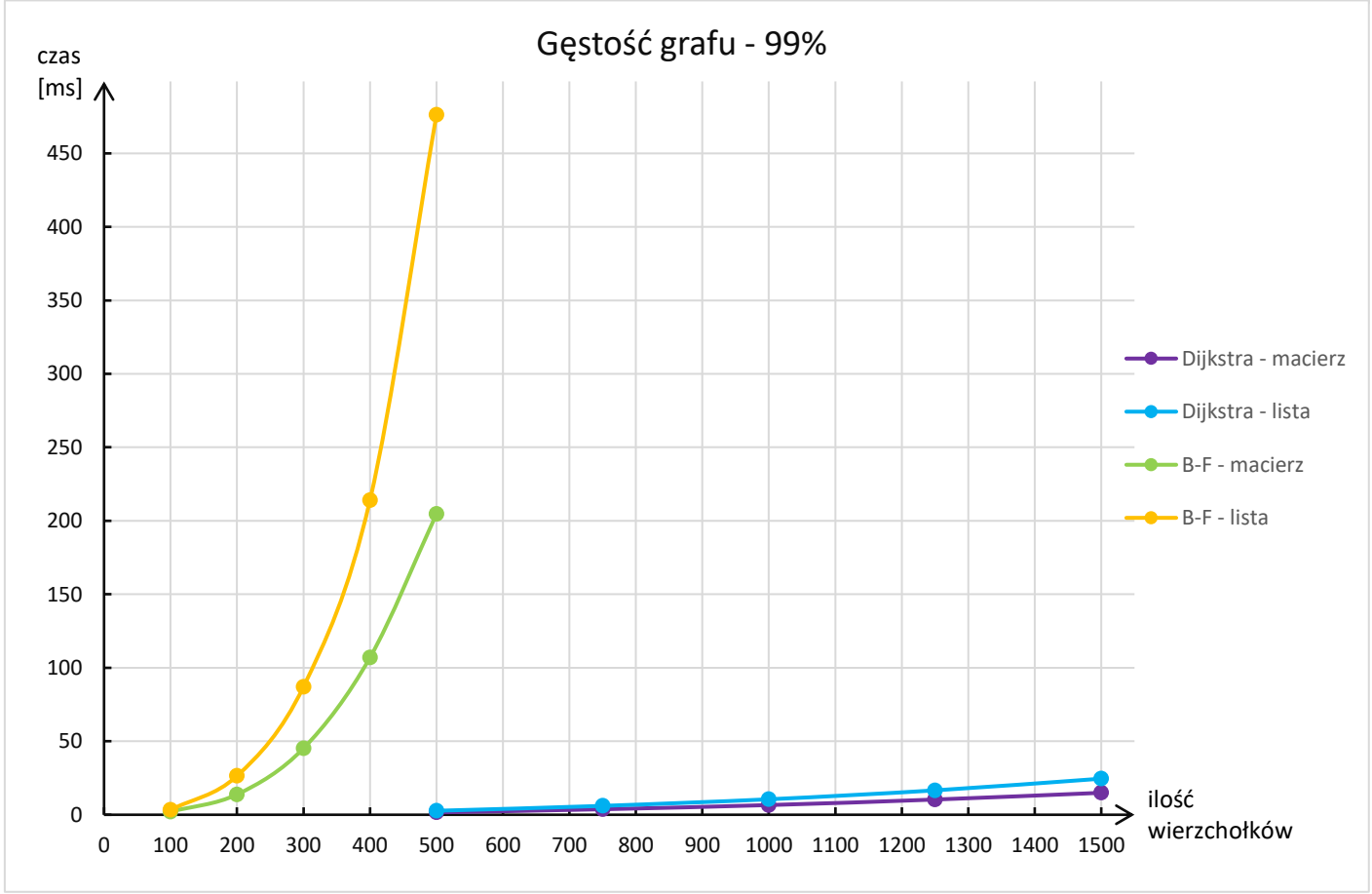

Najkrótsza ścieżka - macierz sąsiedztwa
czas [ms]
800
700
600
500
400
300
200
100
0
0 100 200 300 400 500 600 700 800 900 1000 1100 1200 1300 1400 1500
ilość wierzchołków
Dijkstra - 25%
Dijsktra - 50%
Dijkstra - 75%
Dijkstra - 99%
BF - 25%
BF - 50%
BF - 75%
BF - 99%


Najkrótsza ścieżka - lista sąsiedztwa
czas [ms]
450
400
350
300
250
200
150
100
50
0
0 100 200 300 400 500 600 700 800 900 1000 1100 1200 1300 1400 1500
ilość wierzchołków
Dijkstra - 25%
Dijkstra - 50%
Dijsktra - 75%
Dijkstra - 99%
BF - 25%
BF - 50%
BF - 75%
BF - 99%


Gęstość grafu - 25%
czas [ms]
550
500
450
400
350
300
250
200
150
100
50
0
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
ilość wierzchołków
Dijkstra - macierz
Dijkstra - lista
B-F - macierz
B-F - lista


Gęstość grafu - 50%
czas [ms]
800
700
600
500
400
300
200
100
0
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
ilość wierzchołków
Dijkstra - macierz
Dijkstra - lista
B-F - macierz
B-F - lista


Gęstość grafu - 75%
czas [ms]
550
500
450
400
350
300
250
200
150
100
50
0
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
ilość wierzchołków
Dijkstra - macierz
Dijkstra - lista
B-F - macierz
B-F - lista


Gęstość grafu - 99%
czas [ms]
450
400
350
300
250
200
150
100
50
0
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
ilość wierzchołków
Dijkstra - macierz
Dijkstra - lista
B-F - macierz
B-F - lista

4.4. Wnioski:

W algorytmie Dijkstry dla reprezentacji listowej wykorzystywana jest zaimplementowana kolejka priorytetowa. W przypadku reprezentacji macierzowej zamiast kolejki priorytetowej, zastosowano tablicę przechowującą klucze, w której indeks tablicy odpowiadał indeksowi wierzchołka.

Śledząc przebieg wykresów dla algorytmu Dijkstry można wysunąć wniosek, że sposób jego implementacji dla obu reprezentacji grafu osiągnął złożoności podawane w literaturze. Przyrównując je do wykresów dla algorytmu Prima złożoności te mogą wynosić O(E lgV). Płaski przebieg linii jest skutkiem przyjętego zakresu na osi OY spowodowanego przez duży rozrzut wartości wprowadzony przez algorytm Bellmana – Forda.

Co więcej w przypadku małych gęstości dłuższy czas wykonywania się zajmuje algorytm w reprezentacji macierzowej, a dla większych gęstości ten w reprezentacji listowej. Wytłumaczenie tego faktu jest następujące: w przypadku istnienia małej liczby krawędzi złożoność przeglądania krawędzi grafu powinna być podobna dla obu reprezentacji. Im jednak liczba krawędzi staje się większa, tym przeglądania ich w postaci macierzy bierze górę (w kontekście złożoności czasowej) nad przeglądaniem kolejnych list sąsiadów.

Wykresy dla algorytmu Bellmana – Forda w obu reprezentacjach wykazują dużą złożoność czasową, co jest zgodne z wiedzą teoretyczną, gdzie wynosi ona O(V E), a jeżeli gęstość grafu wynosi w przybliżeniu 100%, czyli $E \approx V^2$, to złożoność ta degeneruje się nawet do $O(V^3)$. Zestawienie otrzymanych wykresów z wykresem funkcji $f(x) = x^3$ wykazuje pewne analogie.

Czas wykonywania się omawianego algorytmu dla reprezentacji listowej jest większy od czasu dla reprezentacji macierzowej dopiero przy gęstości grafu wynoszącej 99%. Zatem dopiero dla gęstości >75% procedura Bellmana – Forda wykonywana na liście sąsiedztwa trwa dłużej, niż ta przeprowadzana na macierzy.

Należy zauważyć, że podobnie, jak przy algorytmie Prima w reprezentacji macierzowej, tak i tu występuje podobna anomalia. Tym razem dotycząca obu procedur znajdowania najkrótszych ścieżek. Na pierwszym wykresie widać, że czasy dla gęstości 50% znów są największe, a te dla 99% najmniejsze. Pozwala to na dalsze wnioskowanie, że zaistniała sytuacja nie jest o tyle skutkiem implementacji tychże algorytmów, co wynika ze sposobu reprezentacji grafu w pamięci komputera (lub jest sumą tych dwóch składowych).

W ostatecznym porównaniu zaimplementowanych algorytmów stwierdzić można, że algorytm Dijkstry jest znacznie szybszy od algorytmu Bellmana – Forda, aczkolwiek ze względu na istnienie kolejki priorytetowej na pewno pod względem złożoności pamięciowej wymaga on większej ilości miejsca. Podczas, gdy drugi algorytm przetwarza jedynie już utworzoną macierz lub listę sąsiadów.

5. Bibliografia:

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Wprowadzenie do algorytmów*, PWN Warszawa 2018, wydanie VII
2. https://algs4.cs.princeton.edu/home/ - R. Sedgewick, *Algorithms*, 4th edition
3. http://eduinf.waw.pl/inf/alg/001_search/0122.php
4. http://antoni.sterna.staff.iiar.pwr.wroc.pl/sdizo/SDiZO_file.pdf
5. http://antoni.sterna.staff.iiar.pwr.wroc.pl/sdizo/SDiZO_time.pdf
6. http://antoni.sterna.staff.iiar.pwr.wroc.pl/sdizo/SDiZO_random.pdf