Struktury danych i złożoność obliczeniowa

Zadanie projektowe nr 2

Badanie efektywności algorytmów grafowych w zależności od rozmiaru instancji oraz sposobu reprezentacji grafu w pamięci komputera

Joanna Zoglowek 264452

Wstęp

W ramach projektu należało zaimplementować określone algorytmy grafowe oraz dokonać pomiarów czasu potrzebnego do ich wykonania w zależności od ilości wierzchołków i gęstości grafu.

Do minimalnych drzew rozpinających należało zaimplementować dwa algorytmy:

- Algorytm Prima oraz Algorytm Kruskala.

Najkrótsze ścieżki w grafie badano za pomocą implementacji algorytmów:

- Algorytmu Dijkstry i Algorytmu Bellmana Forda.

Po zaimplementowaniu algorytmów należało dokonać pomiaru czasu działania algorytmów w zależności od rozmiaru grafu oraz jego gęstości.

Badania były wykonywane 1000 razy dla losowo generowanej liczby wierzchołków oraz gęstości grafu: 25 %, 50 %, 75 % oraz 99 %.

Sposób generowania grafu

- Wygenerowanie losowego minimalnego grafu spójnego i zapisanie go macierzy. Obliczanie ile trzeba dogenerować krawędzi dla danej gęstości grafu i dla danej liczby wierzchołków. Obliczenie z jakim prawdopodobieństwem należy generować krawędź podczas przeglądania macierzy i natrafienia na 0, aby uzyskać zadaną gęstość.
- Dogenerowanie losowe brakującej liczby krawędzi do macierzy.
- Na podstawie macierzy wygenerowanie listy sąsiedztwa.

Sposób zapisu do pliku

- Zapisywanie do pliku z macierzy wszystkich krawędzi (po natrafieniu w macierzy wagi > 0) zapisujemy indeksy jako początek i koniec krawędzi.

Sposób odczytu z pliku

- Wczytanie po kolei wag krawędzi z pliku dla danych początków i końców krawędzi do macierzy i listy.

Opis opcji menu:

1. Opcje dla grafu skierowanego
2. Opcje dla grafu nieskierowanego

Po wybraniu jednej z opcji, czy to dla grafu skierowanego czy
nieskierowanego możemy wybrać jedną z opcji podmenu:

1. Generowanie losowego grafu. Wybranie tej opcji powoduje wygenerowanie losowego grafu i zapisanie go w reprezentacji macierzowej i listowej.
2. Generowanie losowe grafu i zapis do pliku. Wybranie tej opcji powoduje wygenerowanie losowego grafu i zapisanie go w reprezentacji macierzowej i listowej oraz zapisanie grafu do pliku w odpowiednim formacie opisanym w poleceniu do zadania (odpowiednio pliki MST.txt i path.txt).
3. Wczytanie grafu z pliku. Wybranie tej opcji powoduje wczytanie z pliku (użytkownik wprowadza nazwę pliku) grafu i zapisanie go w reprezentacji macierzowej i listowej.

Po wygenerowaniu losowym grafu (opcja 1, 2) lub wczytaniu grafu z pliku (opcja 3) opcje 4-7 pozwalają na testowanie odpowiednich algorytmów dla grafów, gdyż struktury macierzowe i listowe są wtedy wypełnione danymi. Po wybraniu algorytmu Prima, Kruskala, Dijkstry, Bellmana-Forda wyniki zapisywane są do odpowiednich plików: PrimMatrix.txt, PrimList.txt…

8. Opcja "testy automatyczne".
Wybór tej opcji powoduje generowanie odpowiedniej liczby losowych grafów do reprezentacji macierzowej i listowej oraz różnych gęstości grafów, pomiar czasu wykonywania się poszczególnych algorytmów i zapis wyników do plików resultsU.txt i resultsD.txt, w zależności od tego czy wykonujemy testy automatyczne dla grafów skierowanych czy nieskierowanych.

Minimalne drzewa rozpinające:

Algorytm Prima

Algorytm Prima jest algorytmem zachłannym wyszukującym minimalne drzewo rozpinające w nieskierowanym grafie ważonym. Oznacza to znalezienie takiego podzbioru krawędzi, który formuje drzewo zawierające każdy wierzchołek, o najniższej możliwej sumie wag tych krawędzi. Dokonuje tego poprzez dodawanie do drzewa po jednym wierzchołku, a następnie wyszukiwaniu krawędzi o najniższej wadze, która dołączy kolejny wierzchołek do drzewa. Pesymistyczna złożoność obliczeniowa równa jest **O (V^2),** jednak dzięki zastosowaniu kolejki

priorytetowej z krawędzią o najniższej wadze na początku, można uniknąć ciągłego przeszukiwania tablicy krawędzi. Zredukowana złożoność to **(E* logV).**

Algorytm Kruskala

Algorytm Kruskala znajduje minimalne drzewo rozpinające w grafie nieskierowanym poprzez dodawanie krawędzi o najmniejszych wagach, unikając tworzenia cykli.
Działanie algorytmu Kruskala:

1. Sortuj krawędzie grafu nieskierowanego według wag.
2. Twórz pusty zbiór MST (minimalne drzewo rozpinające).
3. Iteruj po posortowanych krawędziach:
    - Jeśli dodanie bieżącej krawędzi do MST nie tworzy cyklu, dodaj ją do MST.
4. Powtarzaj krok 3, dopóki MST nie zawiera wszystkich wierzchołków lub nie przetworzysz wszystkich krawędzi.

Złożoność czasowa algorytmu Kruskala: **O(E log E)**, gdzie E to liczba krawędzi. Złożoność pamięciowa algorytmu Kruskala: **O(V + E),** gdzie V to liczba wierzchołków.

Najkrótsze ścieżki grafu:

Algorytm Dijkstry

Algorytm Dijkstry to algorytm służący do znajdowania najkrótszych ścieżek w grafie ważonym, skierowanym z jednym źródłem. Pozwala znaleźć najkrótszą ścieżkę od danego wierzchołka źródłowego do wszystkich innych wierzchołków grafu.
Działanie algorytmu Dijkstry:

1. Inicjalizuj zbiór odległości distance dla każdego wierzchołka grafu. Ustaw odległość źródłowego wierzchołka na 0, a pozostałe na nieskończoność.
2. Utwórz zbiór odwiedzonych wierzchołków visited.
3. Powtarzaj następujące kroki, dopóki istnieją nieodwiedzone wierzchołki:
    - Wybierz wierzchołek u o najmniejszej odległości, który nie został jeszcze odwiedzony.
    - Oznacz wierzchołek u jako odwiedzony.
    - Dla każdego sąsiedniego wierzchołka v wierzchołka u:
        - Oblicz nową odległość dla wierzchołka v, sumując odległość do wierzchołka u i wagę krawędzi między nimi.

- Jeśli nowa odległość jest mniejsza od dotychczasowej odległości do wierzchołka v, zaktualizuj wartość distance[v].

4. Po zakończeniu algorytmu, distance zawiera najkrótsze odległości od źródłowego wierzchołka do wszystkich innych wierzchołków.

Złożoność czasowa algorytmu Dijkstry zależy od sposobu implementacji kolejki priorytetowej. Najbardziej efektywna jest implementacja przy użyciu kopca binarnego. Ma złożoność **O((V + E) log V)**, gdzie V to liczba wierzchołków, a E to liczba krawędzi. W przypadku używania kopca Fibonacciego, złożoność czasowa wynosi **O(V log V + E)**, co czyni go bardziej wydajnym dla grafów gęstych.

Algorytm Bellmana Forda

Algorytm Bellmana Forda jest algorytmem obliczającym najkrótszą ścieżkę pomiędzy wierzchołkiem startowym, a wszystkimi pozostałymi wierzchołkami w grafie ważonym. W przeciwieństwie do innych tego typu algorytmów radzi sobie z negatywnymi cyklami występującymi w przypadku, gdy krawędzie z ujemnymi wagami mogłyby doprowadzić do wagi ścieżki dążącej do -∞. Skutkuje to jednak dłuższym czasem wykonywania algorytmu. Schemat działania oparty jest na relaksacji krawędzi, w wyniku czego z każdym obiegiem algorytmu (liczba obiegów jest równa liczbie wierzchołków – 1) oszacowanie najkrótszej ścieżki jest dokładnie oszacowanie, aż do uzyskania najlepszego wyniku. Następnie wykonywany jest dodatkowy obieg w wyniku którego możemy ocenić czy istnieje cykl ujemny. W kodzie nie używałam ujemnych wag, żeby lepiej porównać alg. Bellmana-Forda i Dijkstry. Teoretyczna złożoność obliczeniowa **O(E ∗ V).**

Powyższa złożoność wynika z faktu, że w najgorszym wypadku algorytm musi „odwiedzić" wszystkie krawędzie w grafie V razy przy relaksacji.

Pomiary

Minimalne drzewo rozpinające:

Algorytm Prima

Pomiary czasu podane są w mikrosekundach.

	Ilość wierzchołków									
Gęstość [%]	10	20	30	40	50	60	70	80	90	100
25	28	153	495	1168	2247	3881	6148	9105	13173	17953
50	56	155	513	1204	2332	4014	6681	9959	14179	19541
75	122	156	516	1219	2335	4048	6365	9518	13530	18520

99	226	157	512	1212	2320	3999	6229	9239	13125	17911

Tabela 1 – implementacja macierzowa

	Ilość wierzchołków									
Gęstość [%]	10	20	30	40	50	60	70	80	90	100
25	30	88	288	778	1549	2994	4748	7673	11492	16031
50	41	186	727	1913	4091	7450	12766	20811	32413	47033
75	48	292	1119	3116	7013	13963	24803	40587	63133	93986
99	57	406	1672	4822	11084	21900	39110	64833	101905	152463

Tabela 2 – implementacja listowa

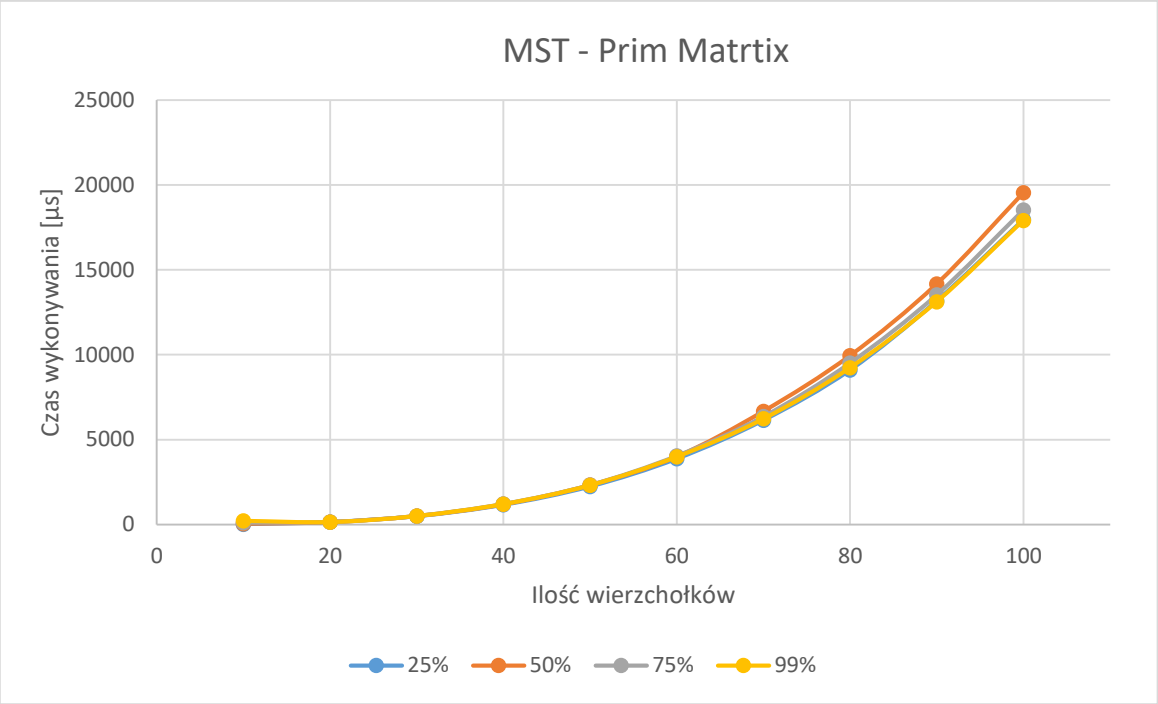
Algorytm Kruskala

	Ilość wierzchołków									
Gęstość [%]	10	20	30	40	50	60	70	80	90	100
25	29	183	820	1980	4791	9160	14631	23927	36275	51545
50	61	189	881	2143	4668	9102	15896	24435	40583	54396
75	113	213	947	2280	4937	8988	16222	25342	37268	54308
99	168	179	674	1722	4446	7782	12917	22395	35558	51586

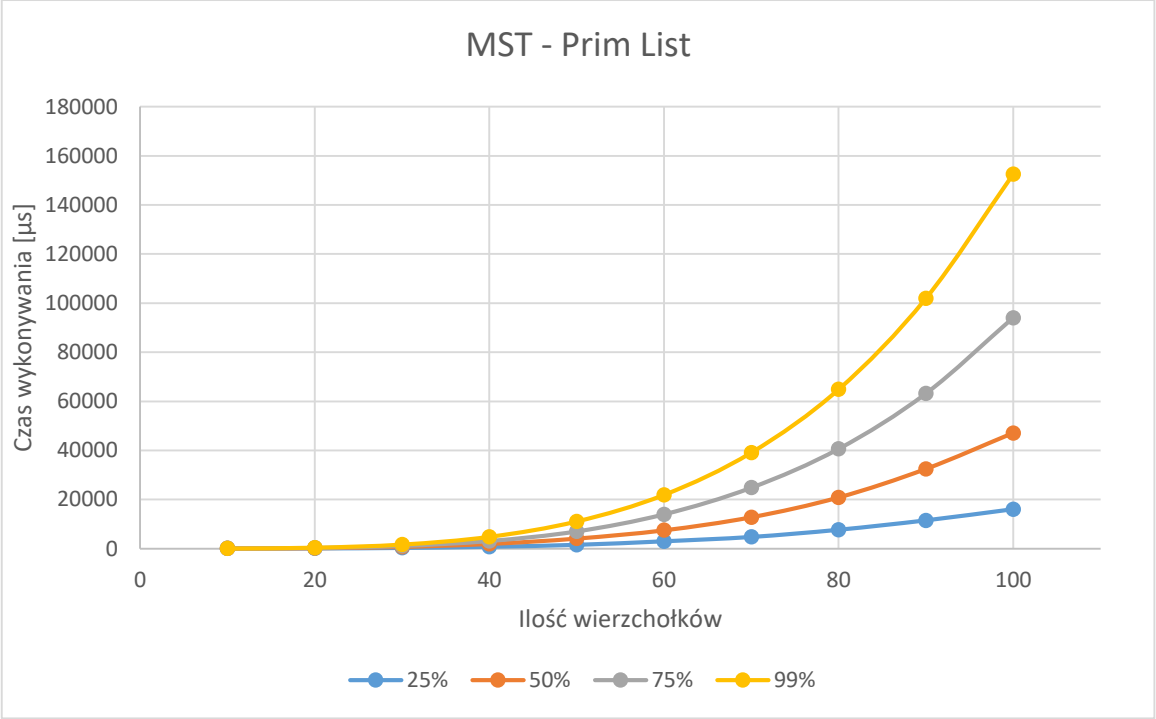
Tabela 3 – implementacja macierzowa

	Ilość wierzchołków									
Gęstość [%]	10	20	30	40	50	60	70	80	90	100
25	64	132	514	1444	2954	5678	10024	15942	24066	35293
50	79	278	1277	3453	7464	14771	26301	43434	67982	101345
75	90	497	2137	6113	14084	28661	51813	86476	135455	204564
99	109	683	3187	9483	22288	44922	82723	138381	218535	333444

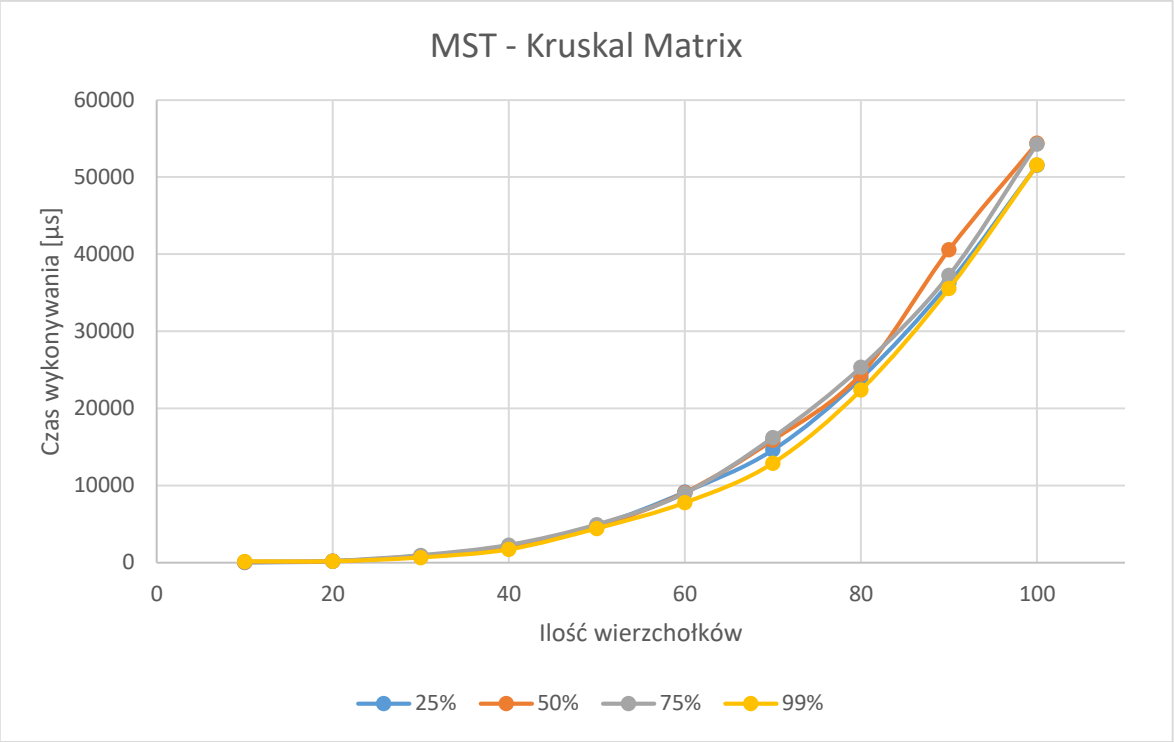
Tabela 4 – implementacja listowa



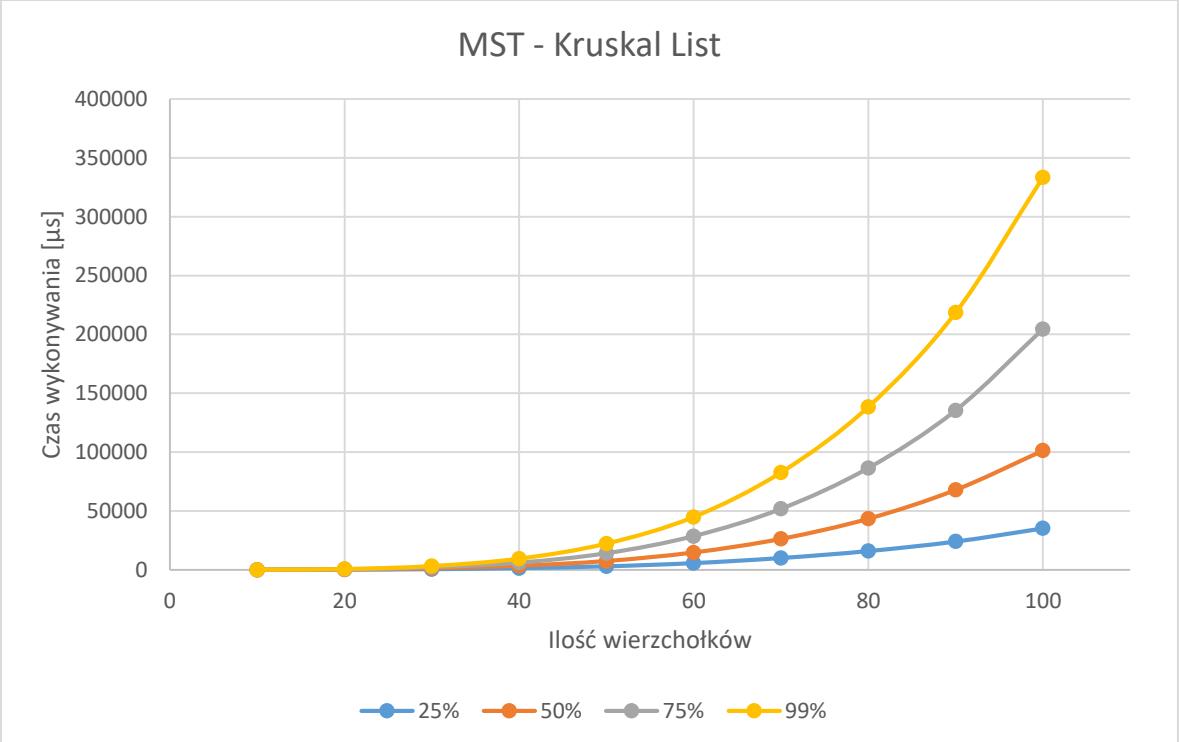
Wykres 1 – Prim w reprezentacji macierzowej



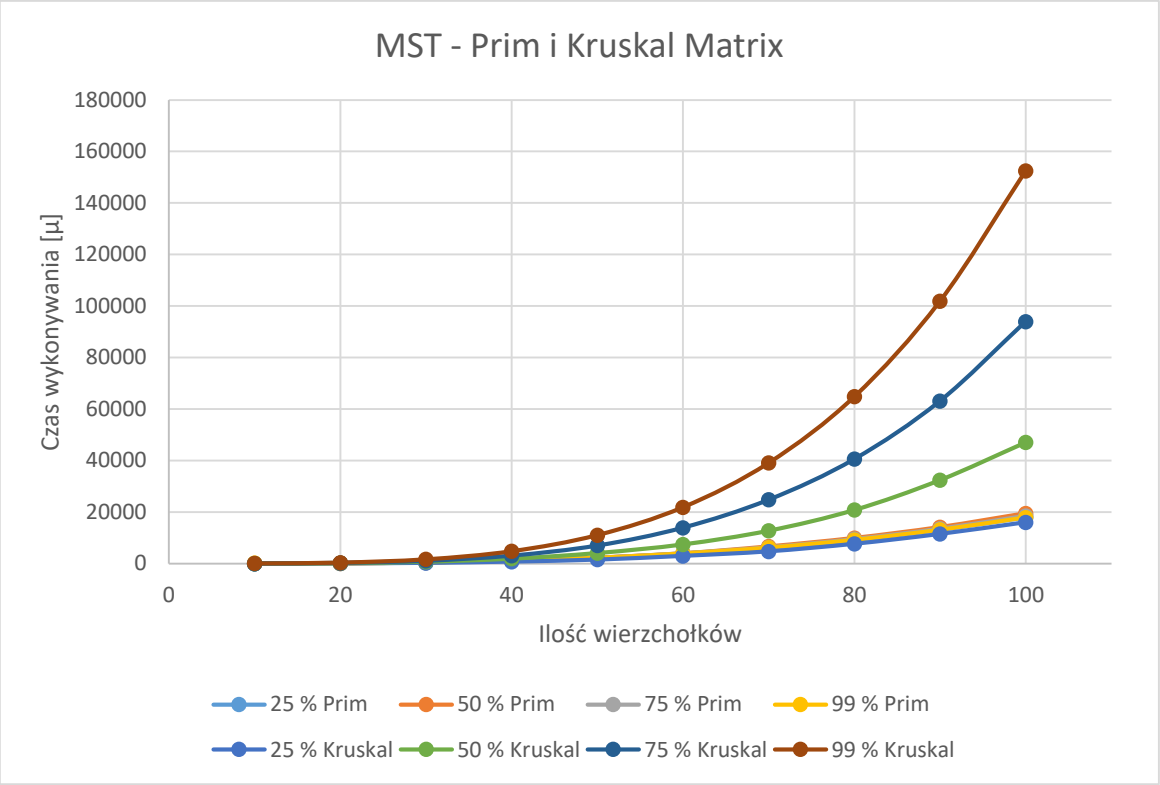
Wykres 2 – Prim w reprezentacji listowej



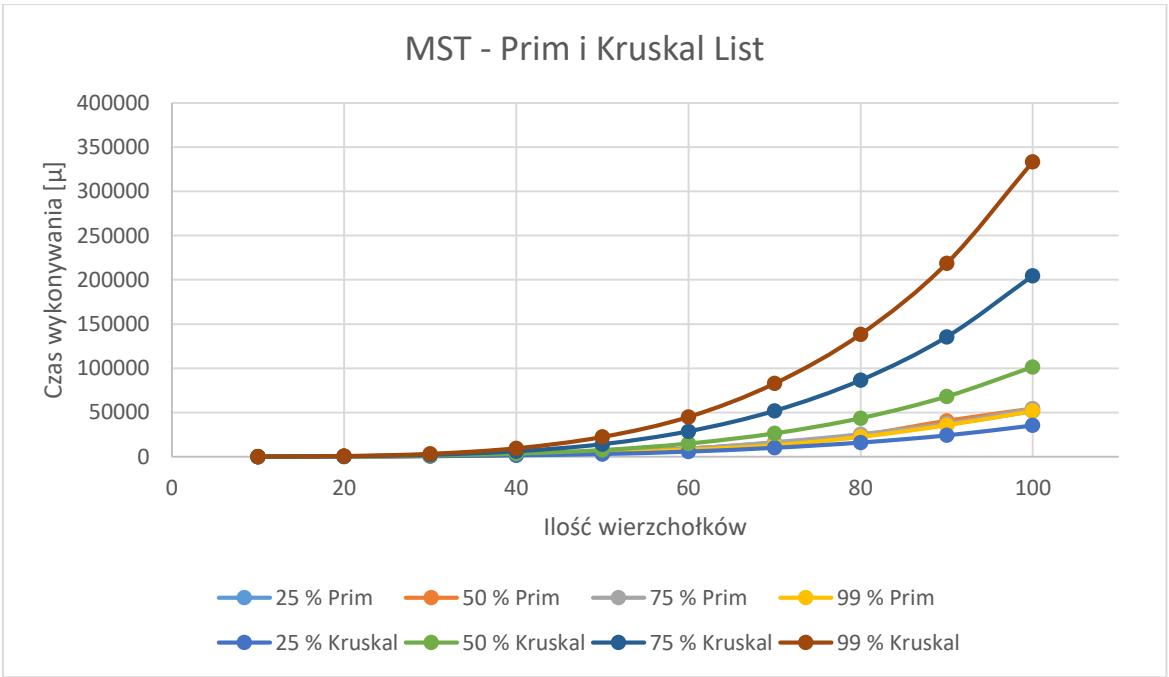
Wykres 3 – Kruskal w implementacji macierzowej



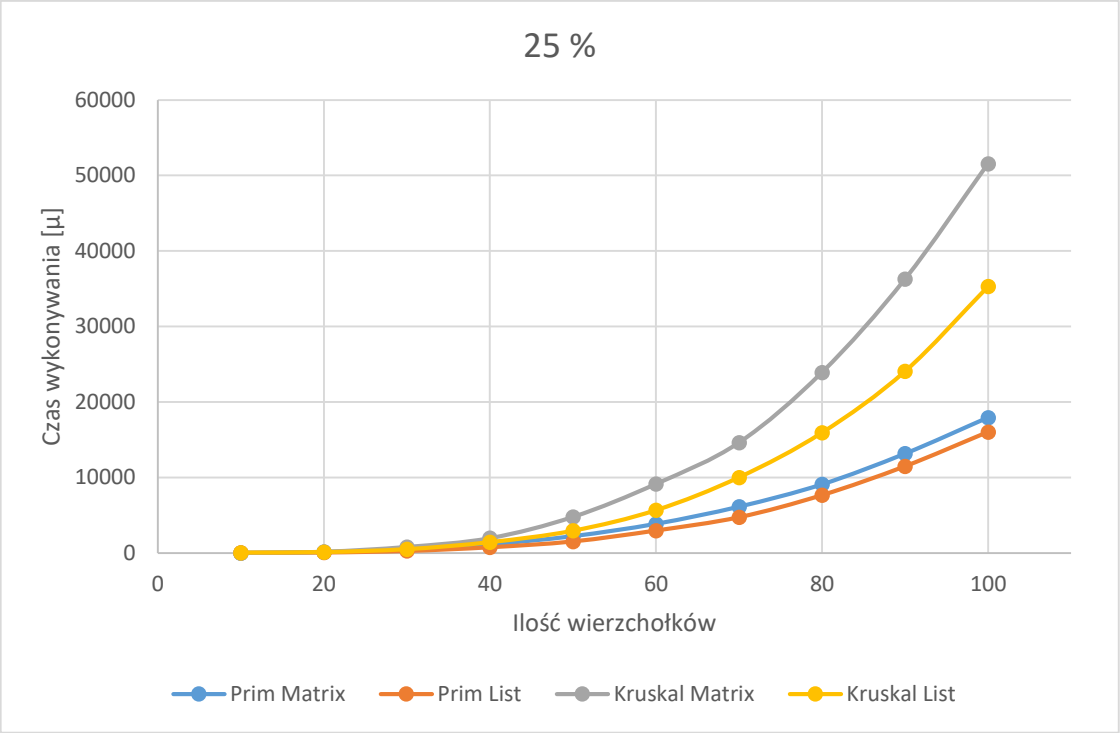
Wykres 4 – Kruskal w reprezentacji listowej



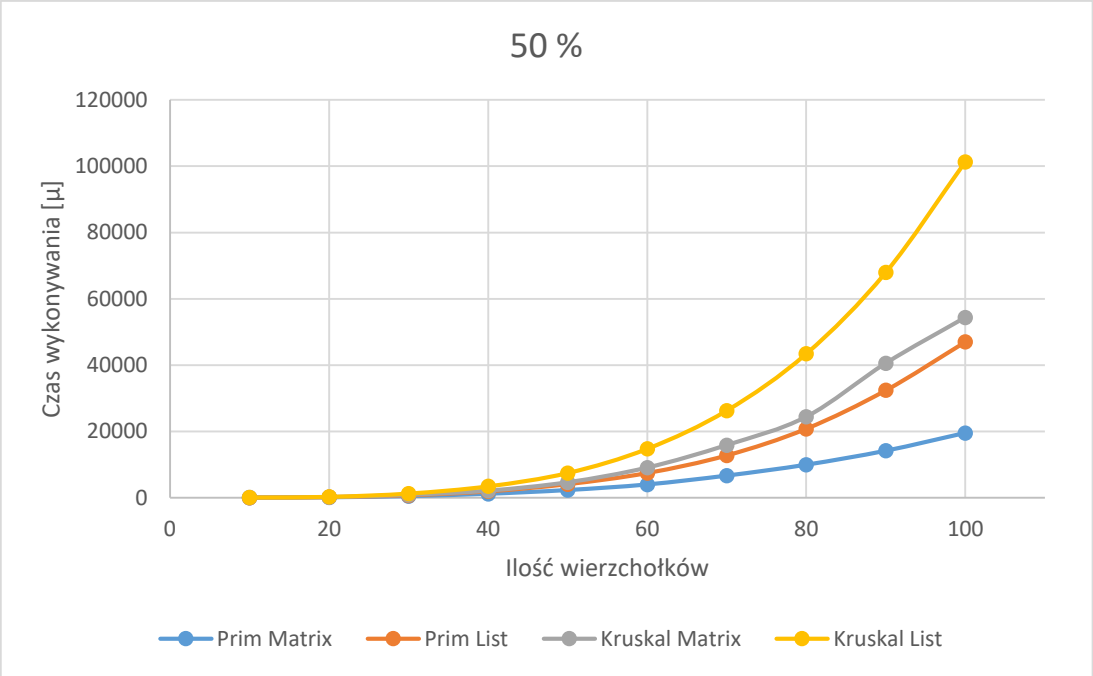
Wykres 5 – Prim i Kruskal w reprezentacji macierzowej



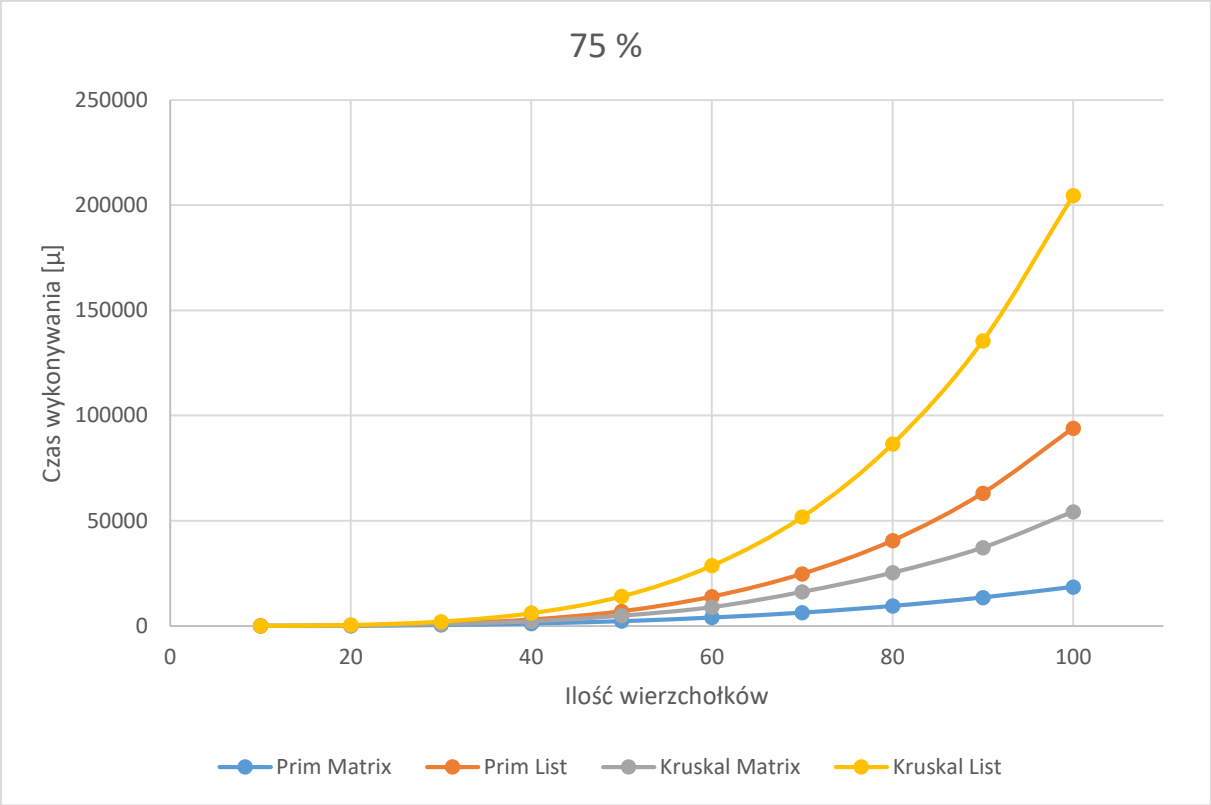
Wykres 6 – Prim i Kruskal w reprezentacji listowej



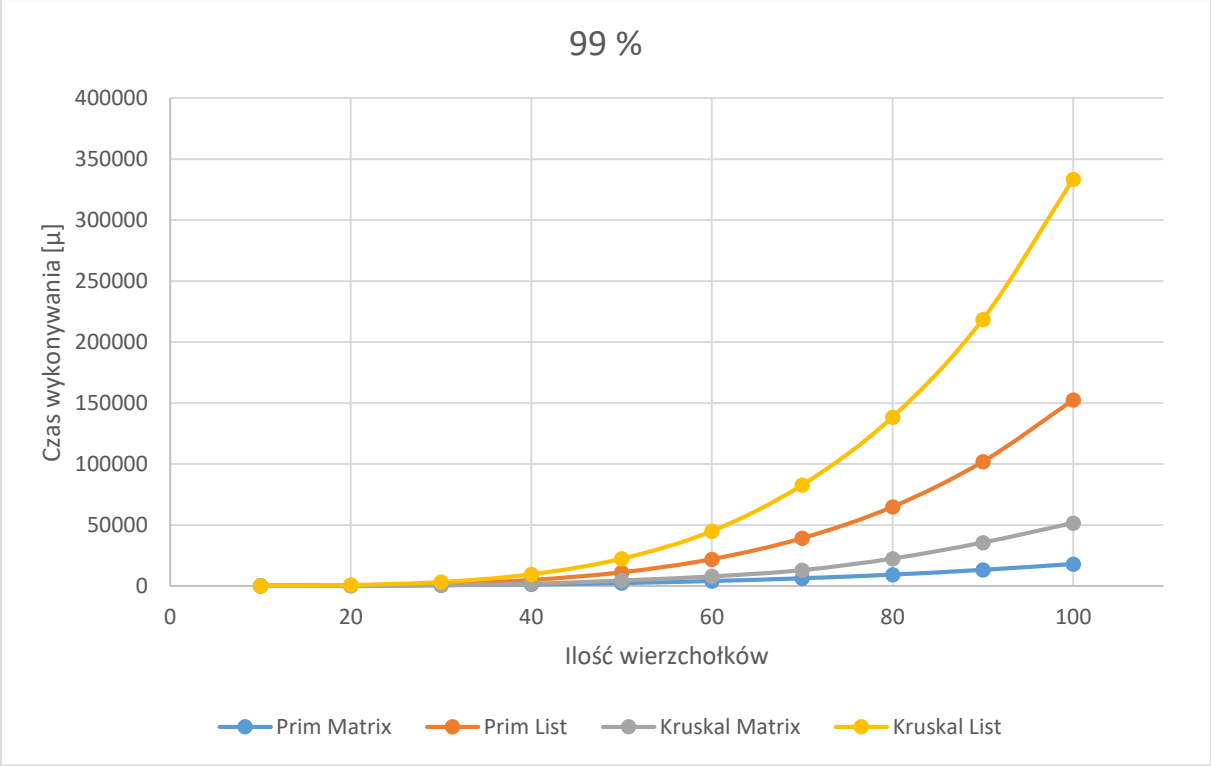
Wykres 7 – Prim i Kruskal przy gęstości 25 %



Wykres 7 – Prim i Kruskal przy gęstości 50 %



Wykres 8 – Prim i Kruskal przy gęstości 75 %



Wykres 9 – Prim i Kruskal przy gęstości 99 %

Najkrótsze ścieżki

Algorytm Dijkstry

	Liczba wierzchołków									
Gęstość	10	20	30	40	50	60	70	80	90	100
25%	12	35	71	142	186	267	362	474	594	740
50%	9	32	72	123	191	270	364	475	600	741
75%	9	34	76	133	193	272	374	486	613	763
99%	10	34	76	126	197	281	374	486	619	761

Tabela 5 – implementacja macierzowa

	Liczba wierzchołków									
Gęstość	10	20	30	40	50	60	70	80	90	100
25%	6	21	46	80	127	178	241	316	404	499
50%	7	26	56	103	157	235	321	432	549	690
75%	8	31	69	125	202	299	418	575	752	970
99%	9	36	85	152	254	430	577	786	1058	1358

Tabela 6 – implementacja listowa

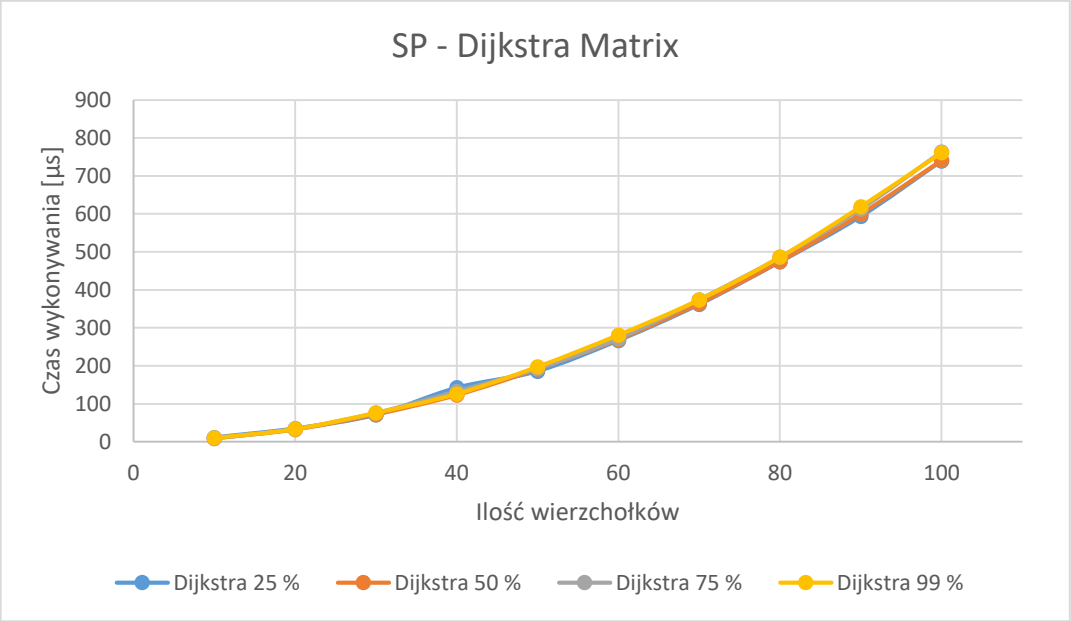
Algorytm Bellmana-Forda

	Liczba wierzchołków									
Gęstość	10	20	30	40	50	60	70	80	90	100
25%	3	10	23	40	77	99	158	270	287	335
50%	4	10	32	57	109	143	247	306	363	554
75%	3	11	32	63	124	246	281	412	520	648
99%	3	15	41	86	183	279	322	496	624	814

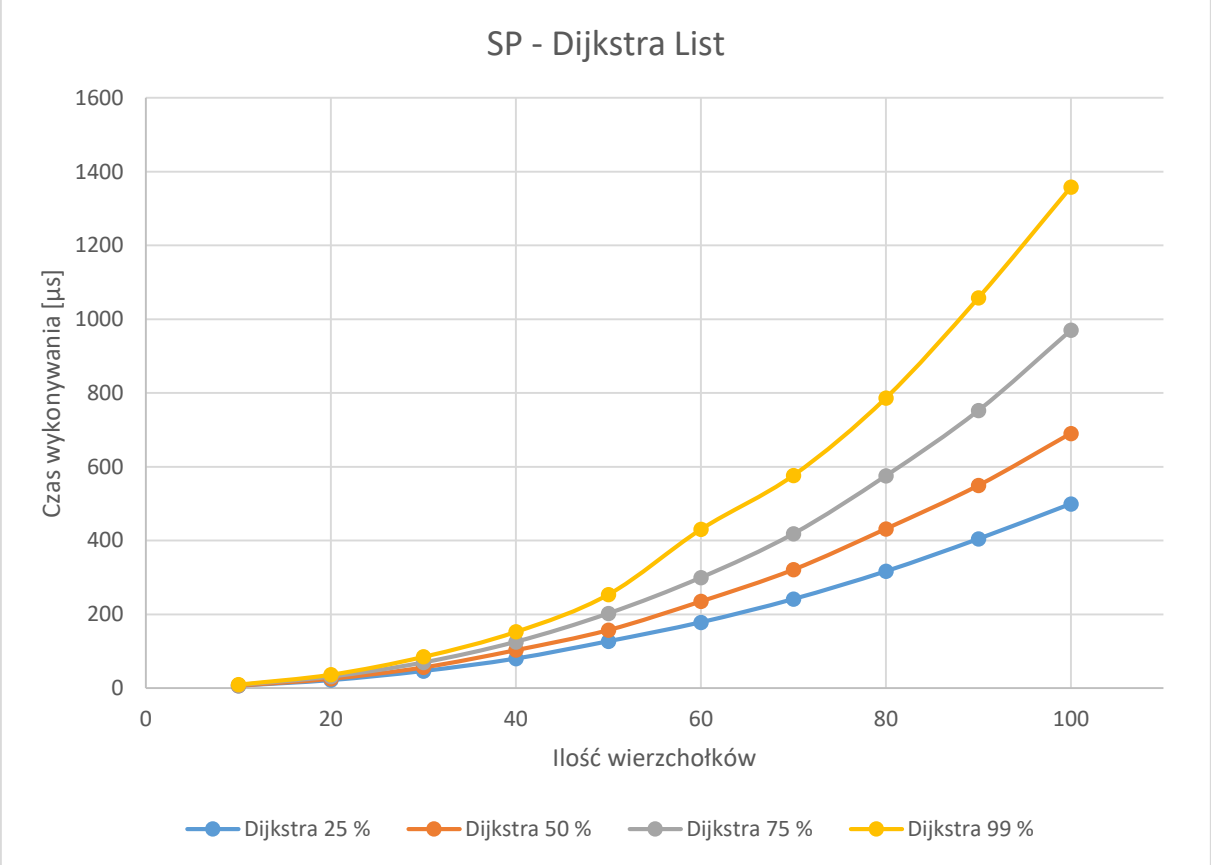
Tabela 7 – implementacja macierzowa

	Liczba wierzchołków									
Gęstość	10	20	30	40	50	60	70	80	90	100
25%	2	10	24	37	57	85	138	161	218	312
50%	5	16	48	79	138	198	286	420	530	709
75%	7	28	67	127	218	376	557	746	1058	1443
99%	9	38	94	185	358	552	828	1254	1769	2307

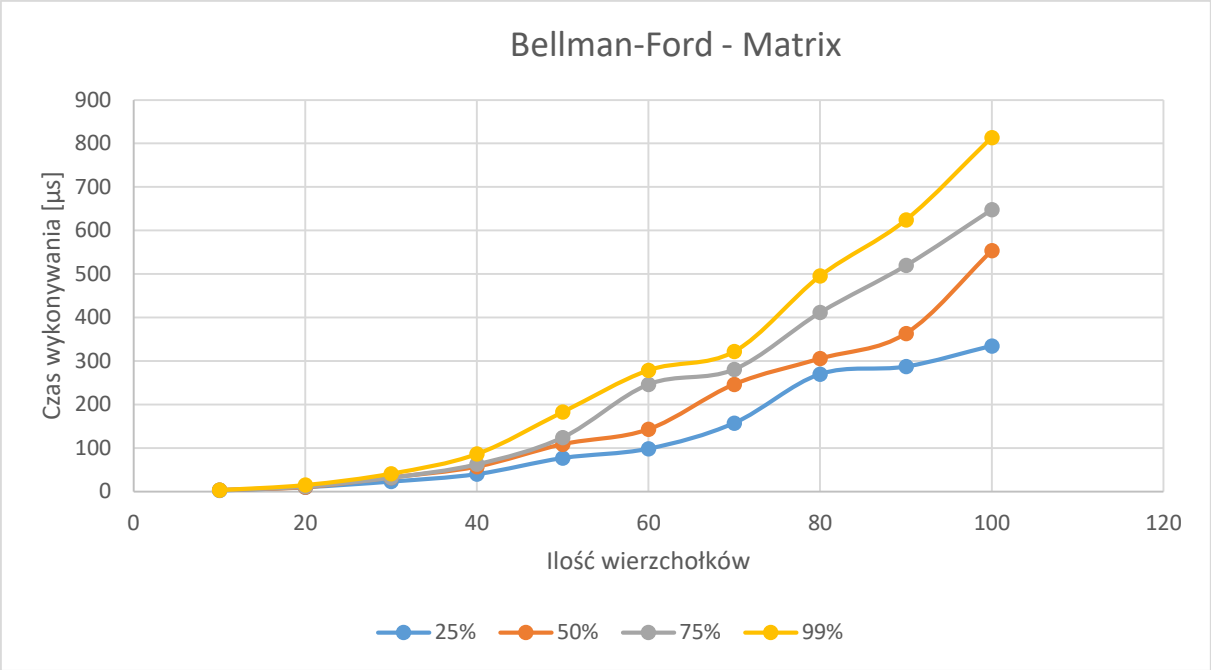
Tabela 8 – implementacja listowa



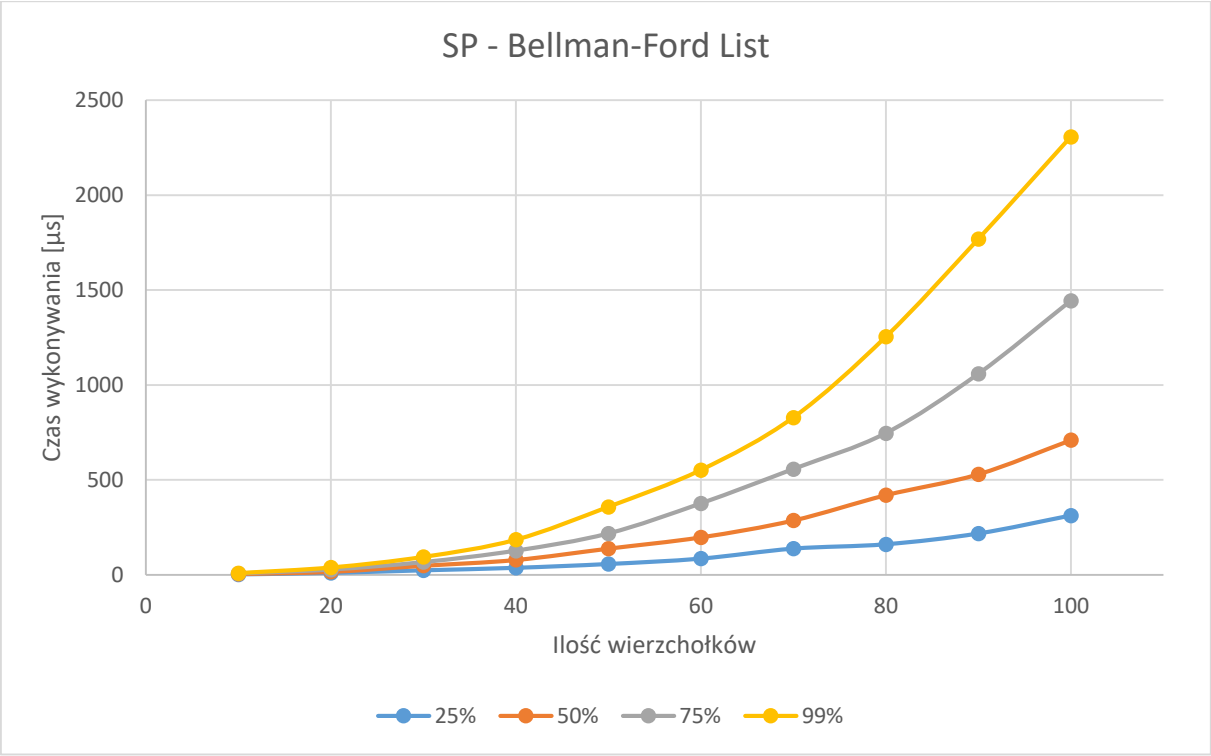
Wykres 10 – Dijkstra w implementacji macierzowej



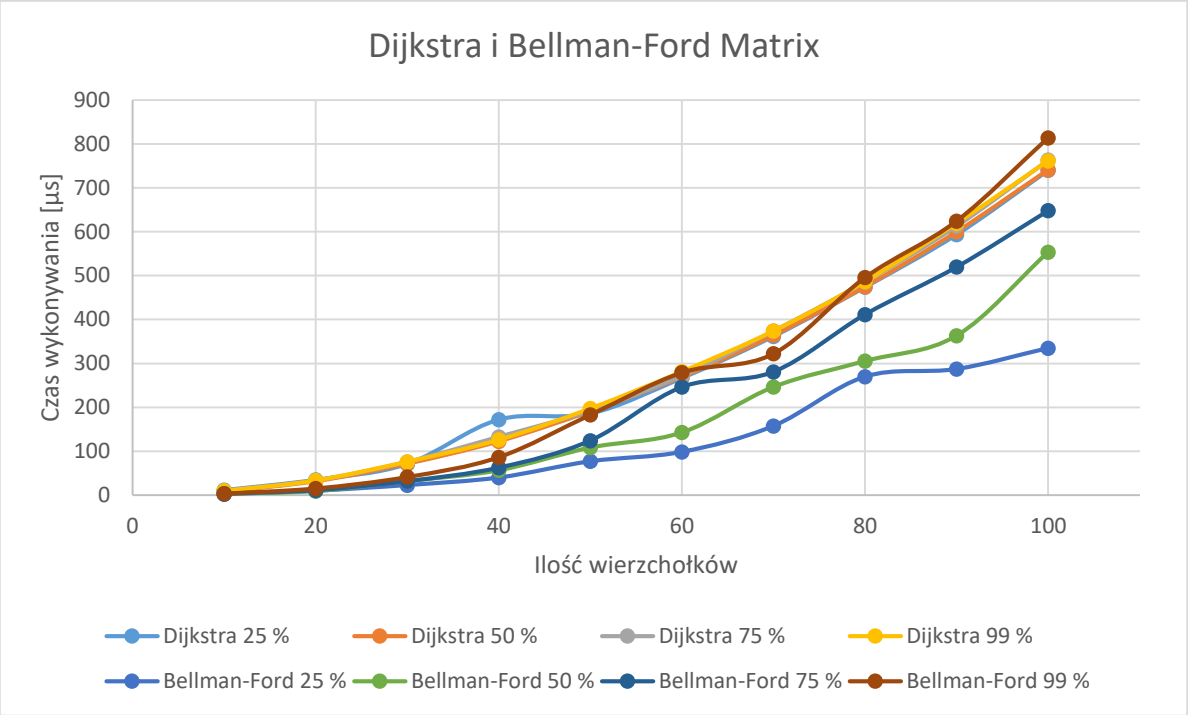
Wykres 11 – Dijkstra w implementacji listowej



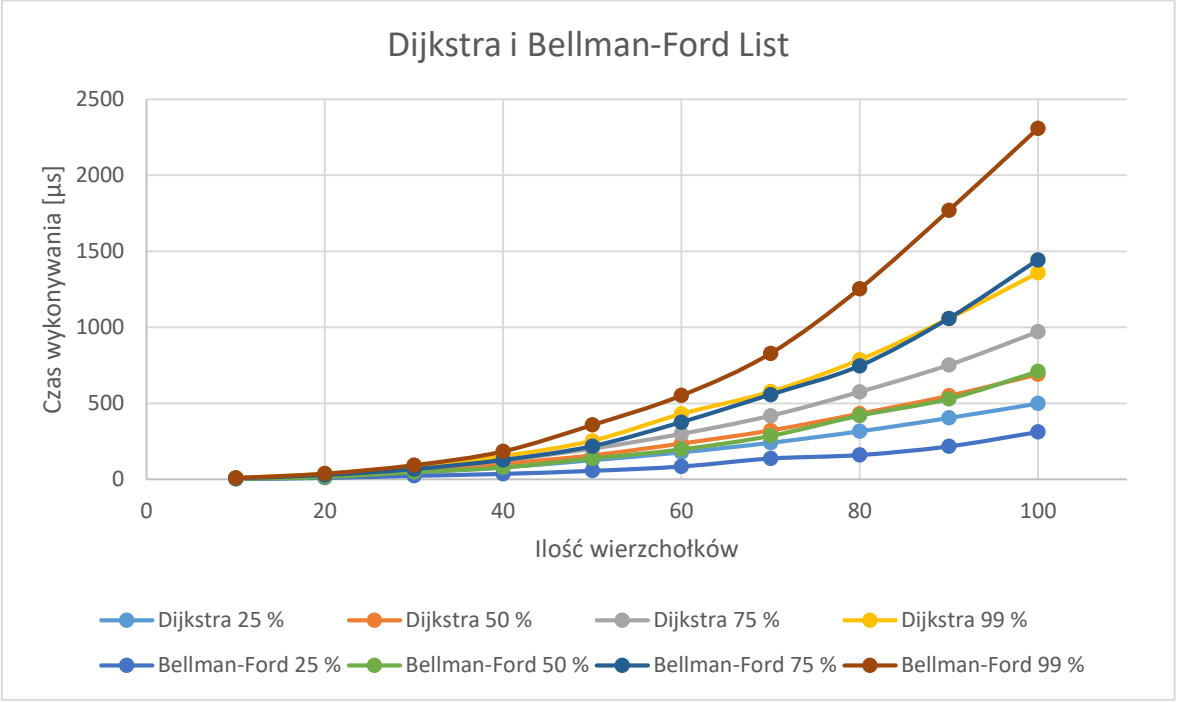
Wykres 12 – Bellman-Ford w implementacji macierzowej



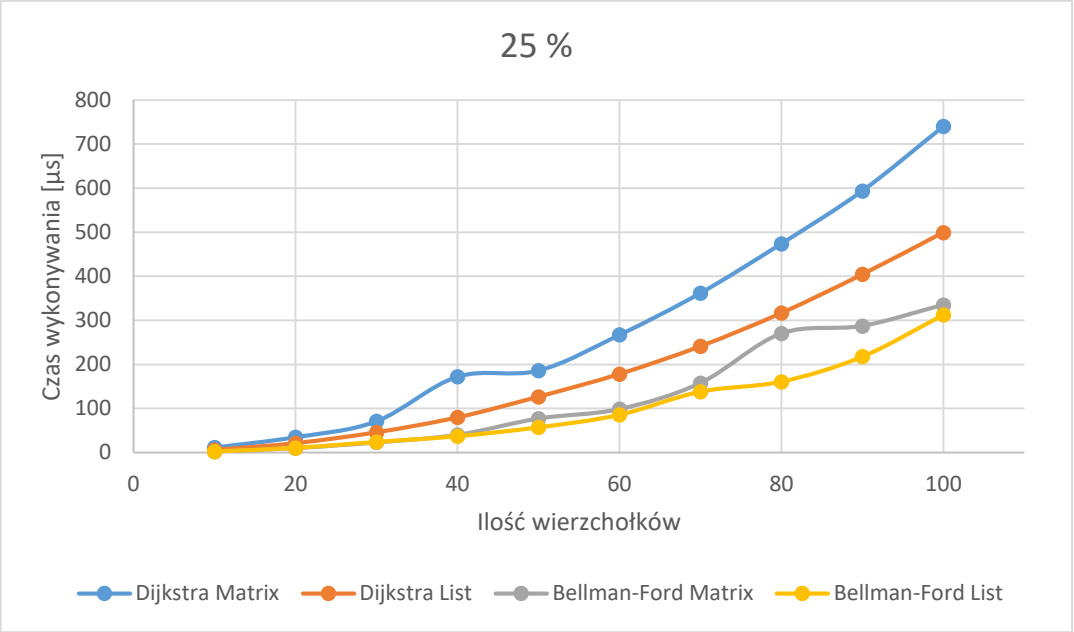
Wykres 13 – Bellman-Ford w implementacji listowej



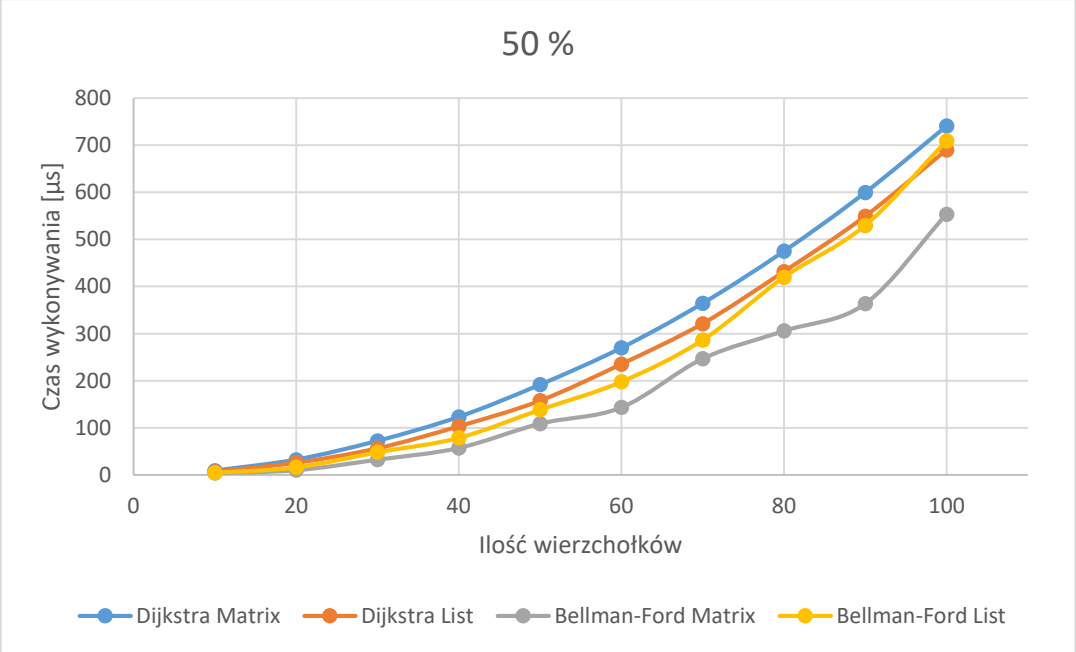
Wykres 14 – Bellman-Ford i Dijkstra w implementacji macierzowej



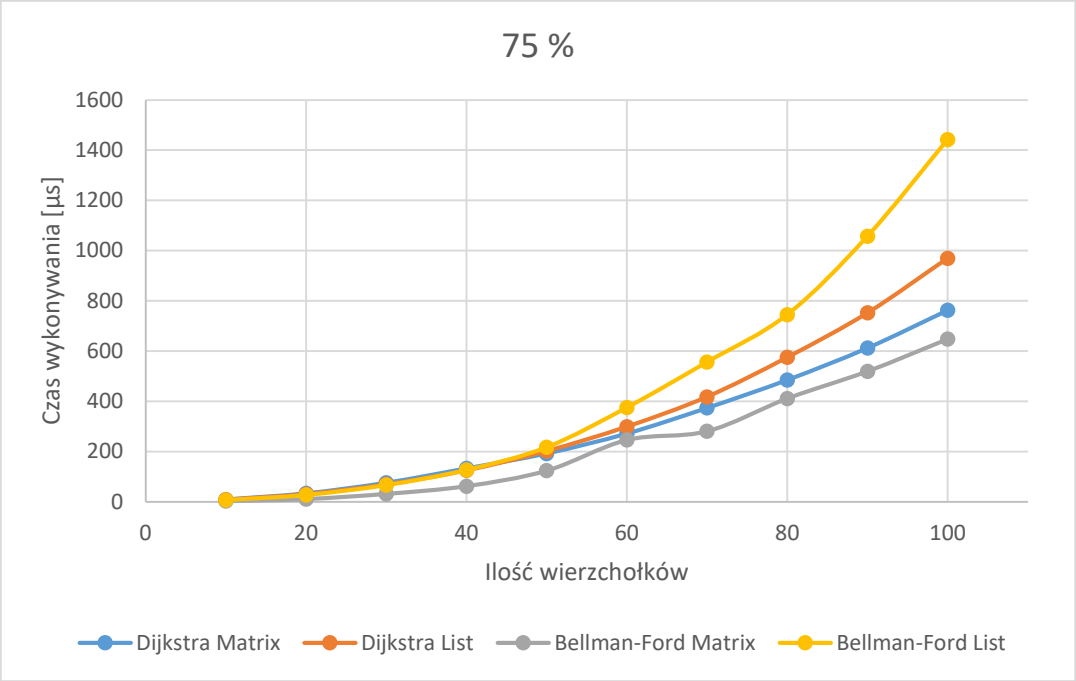
Wykres 15 – Bellman-Ford i Dijkstra w implementacji listowej



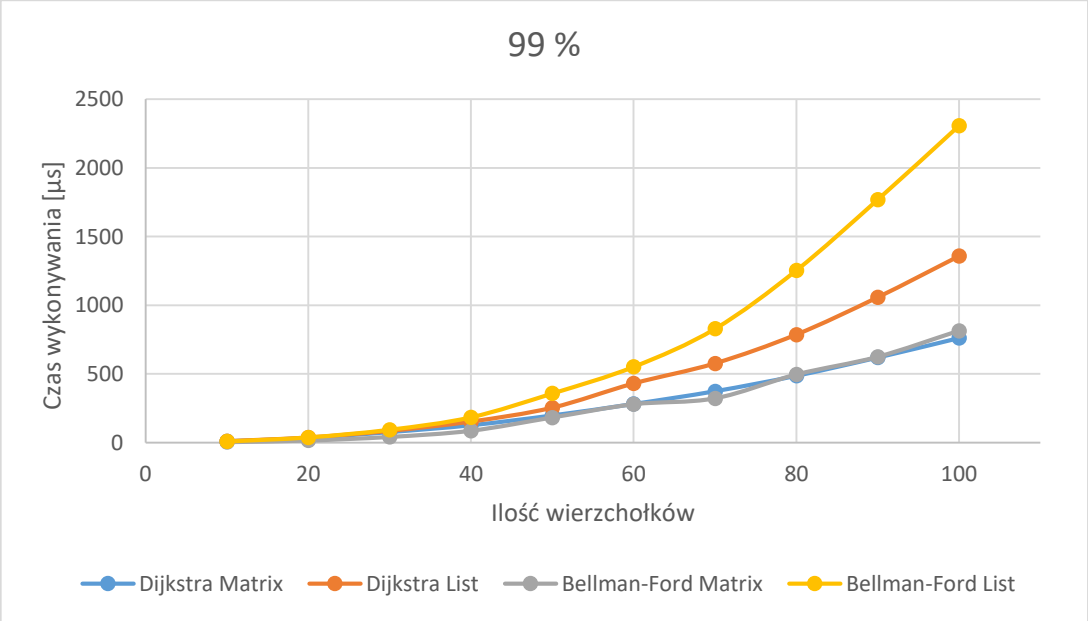
Wykres 16 – Bellman-Ford i Dijkstra przy gęstości 25 %



Wykres 17 – Bellman-Ford i Dijkstra przy gęstości 50 %



Wykres 18 – Bellman-Ford i Dijkstra przy gęstości 75 %



Wykres 19 – Bellman-Ford i Dijkstra przy gęstości 99 %

Wnioski

Na wykresach przedstawiających porównanie czasu trwania algorytmów znajdujących najmniejsze drzewo rozpinające widzimy, że algorytm Prima zaimplementowany na macierzowej reprezentacji grafu jest najszybszy dla prawie wszystkich gęstościach. Jedynie dla grafu o gęstości 25 % implementacja listowa jest nieco szybsza od implementacji macierzowej.

Najwolniejszy okazał się algorytm Kruskala dla implementacji listowej, z wyjątkiem grafu dla gęstości 25 %. Reprezentacja macierzowa jest szybsza dla wszystkich gęstości grafów dla algorytmów Prima i Kruskala.
Do implementacji algorytmu Prima nie korzystałam z kolejki priorytetowej, co powoduje pogorszenie złożoności czasowej algorytmu. Dla reprezentacji macierzowej wykresy złożoności czasowej są nieco gorsze od O (V^2). Dla reprezentacji listowej złożoność czasowa jest gorsza od złożoności czasowej macierzowej.

Czas wykonywania się algorytmu Dijkstry dla implementacji macierzowej jest praktycznie taki sam dla różnych gęstości grafu. Czas wykonywania się algorytmu Dijkstry dla implementacji listowej wydłuża się wraz z gęstością grafu bo wzrastają długości list do przetworzenia.
Czas wykonywania się algorytmu Bellmana-Forda dla implementacji listowej i macierzowej wydłuża się wraz z gęstością grafu.

Dla implementacji macierzowej algorytm Bellmana-Forda jest szybszy od algorytmu Dijkstry dla rzadkich grafów, dla grafu gęstego (99%) algorytm Dijkstry jest szybszy.
Dla implementacji listowej algorytm Bellmana-Forda jest szybszy od algorytmu Dijkstry dla rzadkich grafów a dla gęstych odwrotnie, co zgadza się z teorią

Dla gęstych grafów implementacja macierzowa okazuje się szybsza od implementacji listowej dla obydwu algorytmów (Dijkstry i Bellana-Forda).

Dla moich implementacji algorytmów Dijkstry i Bellmana-Forda, dla reprezentacji macierzowej i listowej wykresy złożoności czasowej są nieco gorsze od O (V^2).

Każdy punkt pomiarowy czasu wykonywania się algorytmu (Prim, Kruskal, Dijkstra, Bellman-Ford) na każdym z wykresów jest uśrednioną wartością dla tysiąca losowo wygenerowanych grafów dla zadanej gęstości grafu i ilości wierzchołków.
Analizując złożoności teoretyczne z uzyskanymi wynikami na wykresach, można stwierdzić, że istnieją pewne rozbieżności. Przyczyny takich różnic mogą wynikać z czynników takich jak: sposób implementacji algorytmów, procesy zachodzące w tle na komputerze podczas działania programu.
Algorytmy listowe przy większej gęstością wykonują się, dłużej ponieważ listy z danymi są znacznie dłuższe.

Wraz ze wzrostem gęstości grafu, obserwujemy wzrost czasu wykonania algorytmów. Istotne jest zatem odpowiednie dostosowanie algorytmów do specyfiki rozwiązywanego problemu i dążenie do ograniczenia gęstości grafu.