2024

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

ROBERT PYTEL (272931)

Spis treści

Wstęp teoretyczny

Algorytm Prima

Ogólne informacje

Metoda ta służy do wyznaczenia minimalnego drzewa rozpinającego (ang. *Minimal Spanning Tree*) w grafie nieskierowanym. Minimalne drzewo rozpinające to podzbiór grafu, który łączy wszystkie wierzchołki bez cykli i o minimalnej sumie wag krawędzi. Algorytm został wynaleziony przez czeskiego matematyka Vojtěcha Jarníka w 1930 roku, a następnie odkryty na nowo przez informatyka Roberta C. Prima w 1957. Dwa lata później niezależnie odkrył go również Edsgr Dijkstra , dlatego też algorytm jest nazywany algorytmem Dijkstry-Prima, algorytmem Jarníka lub algorytmem Prima- Jarníka.

Opis działania

Na początku algorytmu wybieramy wierzchołek startowy i dodajemy go do zbioru wierzchołków rozpatrzonych. Pozostałe wierzchołki umieszczamy w zbiorze wierzchołków nierozpatrzonych. Następnie tworzymy listę wszystkich krawędzi łączących zbiór wierzchołków rozpatrzonych ze zbiorem wierzchołków nierozpatrzonych. Z utworzonej listy wybieramy krawędź o najmniejszej wadze i dodajemy ją do zbioru krawędzi minimalnego drzewa rozpinającego. Wierzchołek końcowy tej krawędzi przenoszony jest ze zbioru wierzchołków nierozpatrzonych do zbioru wierzchołków rozpatrzonych. Algorytm kończy swoje działanie w momencie gdy zbiór wierzchołków rozpatrzonych zawiera wszystkie wierzchołki grafu. W przeciwnym wypadku powtarzamy krok związany z tworzeniem listy krawędzi i wybieraniem tej o minimalnej wadze.

Analiza złożoności obliczeniowej

Złożoność obliczeniowa algorytmu zależy od jego implementacji. W przypadku gdy do zaimplementowania kolejki, która będzie przechowywała dane o krawędziach i ich wagach użyjemy kopca binarnego, to złożoność algorytmu będzie rzędu $O(E * \log(V))$, gdzie E to liczba krawędzi grafu natomiast V to liczba wierzchołków grafu. W przypadku użycia kopca Fibonacciego uzyskujemy średnio najlepszą wydajność równą $O(E + V\log(V))$.

Algorytm Kruskala

Ogólne informacje

Kolejny algorytm z grupy algorytmów wyznaczających minimalne drzewo spinającego grafu. Podobnie jak algorytm Prima należy on do kategorii algorytmów zachłannych. Jego pierwsza publikacja miała miejsce w 1956 roku, której autorem był amerykański matematyk Joseph Kruskal. Publikacja miała miejsce w czasopiśmie *Proceedings of the American Mathematical Society*.

Opis działania

Ogólna idea algorytmu Kruskala opiera się na operacji dodawania krawędzi do drzewa MST pod warunkiem, że dodanie krawędzi nie spowoduje powstania cyklu w podzbiorze. Pierwszym krokiem utworzenie zbioru posortowanych rosnąco krawędzi grafu według wagi. Kolejnym aspektem, który należy uwzględnić w przypadku tego algorytmu jest podział wierzchołków na grupy. Na początku liczba grup jest równa ilości wierzchołków. W momencie dodania krawędzi do grafu, wierzchołki tej krawędzi otrzymują ten sam identyfikator grupy. Dzięki temu można stwierdzić czy dodanie określonej krawędzi spowoduje powstanie cyklu. Taka sytuacja będzie miała miejsce gdy wierzchołki krawędzi, którą chcemy dopiero dodać, będą w tej samej grupie. W takim przypadku nie możemy dodać krawędzi do MST.

Analiza złożoności obliczeniowej

Złożoność obliczeniowa zależy od implementacji algorytmu. W programie została zaimplementowana wersja lasu drzew rozłącznych z kompresją ścieżki. W tym przypadku złożoność obliczeniowa fazy algorytmu odpowiedzialnej za wyznaczenie drzewa MST jest rzędu $O(E\alpha(E,V))$, gdzie α jest niezwykle wolno rosnącą funkcją, odwrotnością funkcji Ackermanna. W takim scenariuszu całkowita złożoność algorytmu będzie zależała od złożoności obliczeniowej algorytmu sortowania użytego do posortowania zbioru krawędzi. W przypadku projektu został użyty algorytm sortowania szybkiego, zatem złożoność obliczeniowa wynosi $O(ElogE)$.

Algorytm Dijkstry

Ogólne informacje

Druga grupa algorytmów dotyczy problemu znajdowania najkrótszej ścieżki w grafie. Zadaniem algorytmu jest wyznaczenie trasy oraz jej całkowitego kosztu pomiędzy wierzchołkiem startowym a pozostałymi wierzchołkami. Warunkiem poprawnego działania algorytmu Dijkstry jest brak wag o ujemnej wartości. Algorytm został opracowany przez informatyka Edsgera Dijkstrę w 1956 roku a następnie opublikowany w 1959 roku.

Opis działania

Działanie algorytmu opiera się na kolejce priorytetowej wszystkich wierzchołków grafu. Priorytetem tej kolejki jest aktualnie wyliczona odległość od wierzchołka startowego. Algorytm działa do momentu gdy kolejka zawiera wierzchołki. W każdym kroku wybieramy z kolejki wierzchołek o najmniejszej odległości od wierzchołka startowego. Dla sąsiadów danego wierzchołka sprawdzamy czy możemy dokonać tak zwanej relaksacji krawędzi, która polega na tym, że obliczamy możliwość dotarcia do danego wierzchołka mniejszym kosztem. Jeżeli taka możliwość istnieje to odpowiednio uaktualniamy dane algorytmu. W momencie zakończenia działania algorytmu otrzymujemy informację o najkrótszych trasach do wierzchołka startowego oraz ich kosztach.

Analiza złożoności obliczeniowej

Złożoność algorytmu zależy od implementacji kolejki priorytetowej. W przypadku zastosowania „naiwnego", w którym wykorzystujemy zwykłą tablicę, otrzymujemy algorytm o złożoności $O(V^2)$, gdzie V oznacza liczbę wierzchołków. Zdecydowanie lepszy wynik zostanie uzyskany w przypadku zaimplementowania kolejki priorytetowej za pomocą kopca binarnego. W takim scenariuszu złożoność wynosi $O(ElogV)$. W przypadku implementacji za pomocą kopca Fibonacciego otrzymujemy złożoność $O(E + VlogV)$. Złożoności te zależne są jednak od gęstości grafu. W przypadku grafów gęstych, czyli takich, w których występuje dużo krawędzi naiwna wersja grafu może okazać się lepszym rozwiązaniem. W przypadku gdy mamy do czynienia z rzadkimi grafami, dobre wyniki uzyskuje druga implementacja.

Algorytm Bellmana-Forda

Ogólne informacje

Drugi algorytm z grupy algorytmów wyszukiwania najkrótszych ścieżek w grafie to algorytm Bellmana-Forda. Podobnie jak algorytm Dijkstry, służy on do wyszukania najkrótszej ścieżki pomiędzy wierzchołkiem startowym a pozostałym wierzchołkami grafu. W przeciwieństwie do algorytmu Dijkstry, w tej wersji algorytmu mogą występować krawędzie o ujemnych wagach, jednakże nie mogą występować cykle o łącznej ujemnej wadze osiągalne ze źródła.

Opis działania

Działanie algorytmu opiera się na wykonaniu $V - 1$ relaksacji każdej krawędzi, która podobnie jak w przypadku algorytmu Dijkstry polega na znalezieniu lepszej trasy dotarcia do danego wierzchołka oraz uaktualnieniu odpowiednich wpisów wykorzystywanych przez algorytm. Nie zawsze konieczne wykonanie jest $V - 1$ iteracji. Jeżeli podczas iteracji nie dokonaliśmy żadnej relaksacji to można zakończyć działanie algorytmu.

Analiza złożoności obliczeniowej

Złożoność algorytmu jest gorsza niż dla algorytmu Dijkstry. Metoda wyszukiwania najkrótszej ścieżki Bellana-Forda ma złożoność $O(V * E)$.

Plan eksperymentu

Celem eksperymentu jest porównanie czasów wykonania poszczególnych algorytmów w zależności od ilości wierzchołków, gęstości oraz reprezentacji pamięciowej grafu. Program został napisany w języku C++.

Struktury danych

Lista sąsiedztwa została zaimplementowana jako lista następników. Ma to znaczenie w przypadku grafów skierowanych. Oznacza to, że dla każdego wierzchołka przechowywana jest lista wierzchołków, które są bezpośrednimi sąsiadami. Co ważne wierzchołki te są końcem krawędzi. W przypadku gdy graf nie jest skierowany dane te nie mają znaczenia i są powielone. Każdy element listy wierzchołków zawiera pole wskaźnikowe na następny element w liście. Struktura jest dynamicznie alokowana na stercie.

W przypadku macierzy incydencji sytuacja nie wymagała tworzenia własnej struktury danych. W programie wykorzystano tablicę wskaźników o rozmiarze VxE, gdzie V to liczba wierzchołków a E to liczba krawędzi. Macierz incydencji jest dynamicznie alokowana na stercie.

Sposób generowania struktur danych

W przypadku listy sąsiedztwa dodawanie nowych elementów ma stały czas. Złożoność tą można uzyskać dzięki dodawaniu elementu na początek listy. Dzięki temu unikamy przeszukiwania całej listy w celu znalezienia jej końca. Dla macierzy incydencji niestety sytuacja jest gorsza. Za każdym razem gdy dodawana jest nowa krawędź wymagane jest utworzenie nowej tablicy wskaźników co wymaga skopiowania poprzedniej struktury danych oraz dodanie informacji dla nowej krawędzi.

Sposób generowania grafu

W celu zapewnienia spójności grafu przyjęto następującą taktykę. Na początku zbiór krawędzi grafu jest pusty. Losowo wybierany jest jeden wierzchołek, dla którego dodajemy $V - 1$ krawędzi, każda do innego wierzchołka. W wyniku tego działania otrzymuje się drzewo rozpinające grafu. Następnym krokiem jest wybranie dwóch losowych wierzchołków i dodanie krawędzi pomiędzy nimi o ile taka już nie istniała. Działanie to jest wykonywane tak długo aż do osiągnięcia zamierzonej gęstości grafu. Dzięki temu uzyskany graf jest spójny oraz zawiera wymaganą ilość krawędzi.

Założenia dla eksperymentu

Pomiary wykonano dla 7 reprezentatywnych liczb wierzchołków grafu: 25, 50, 75, 100, 125, 150, 175. Gęstości grafu, które przebadano to 25%, 50% oraz 99%. W celu uzyskania małego błędu pomiarowego każdy pomiar został wykonany 50 razy. Uzyskane wyniki są uśrednionym czasem 50 iteracji. Do pomiaru czasu wykonania algorytmu użyto funkcji std::chrono::high_resolution_clock.

Wyniki

Poniższe tabele zawierają wyniki uzyskanych pomiarów. Tabele zostały podzielone względem kilku kategorii: problem, który był rozwiązywany oraz sposób reprezentacji grafu w pamięci. Kolumna opisana jest za pomocą symbolu: nazwa algorytmu oraz liczba oznaczają gęstość grafu w procentach.

Tabela 1 Czasy działania algorytmów minimalnego drzewa rozpinającego w mikrosekundach dla grafu w postaci listy sąsiedztwa

Ilość wierzchołkow	PRIM25 [µs]	KRUSKAL25 [µs]	PRIM50 [µs]	KRUSKAL50 [µs]	PRIM99 [µs]	KRUSKAL99 [µs]
25	6,726	7,866	8,844	13,754	12,436	25,236
50	20,648	29,448	25,512	56,454	42,006	117,54
75	36,06	68,928	56,32	135,964	98,76	271,302
100	54,256	111,058	133,962	240,468	426,986	729,262
125	130,528	196,03	355,948	541,496	732,978	1220,4
150	241,994	314,05	528,518	860,354	1016,25	2028,27
175	365,575	670,305	700,535	1160,28	1281,46	2770,37

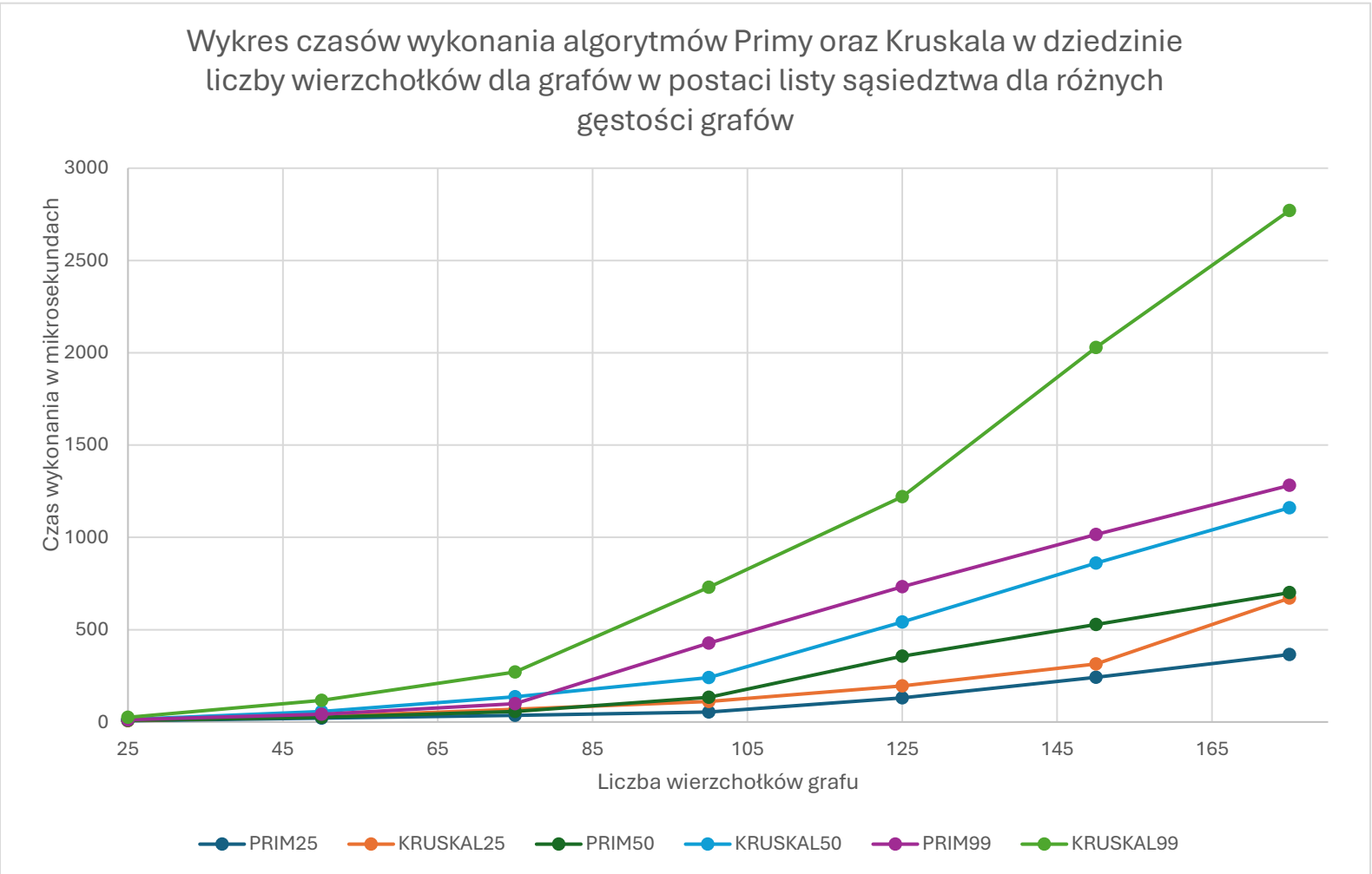


Tabela 2 Czasy działania algorytmów minimalnego drzewa rozpinającego w mikrosekundach dla grafu w postaci macierzy incydencji

Ilość wierzchołkow	PRIM25 [µs]	KRUSKAL25 [µs]	PRIM50 [µs]	KRUSKAL50 [µs]	PRIM99 [µs]	KRUSKAL99 [µs]
25	49,122	11,594	102,112	22,652	181,194	42,488
50	293,902	61,488	450,54	118,954	914,084	244,156
75	757,46	163,006	1174,41	353,222	2414,15	682,416
100	1133,82	345,058	2382,89	721,334	8301,46	1756,28
125	2035,72	645,126	6411,25	1769,9	18970,8	3999,83
150	3804,57	1043,02	12537,4	2821,52	34271,5	6782,34
175	9234,46	2768,99	21449,3	4311,22	49768,4	10534,7

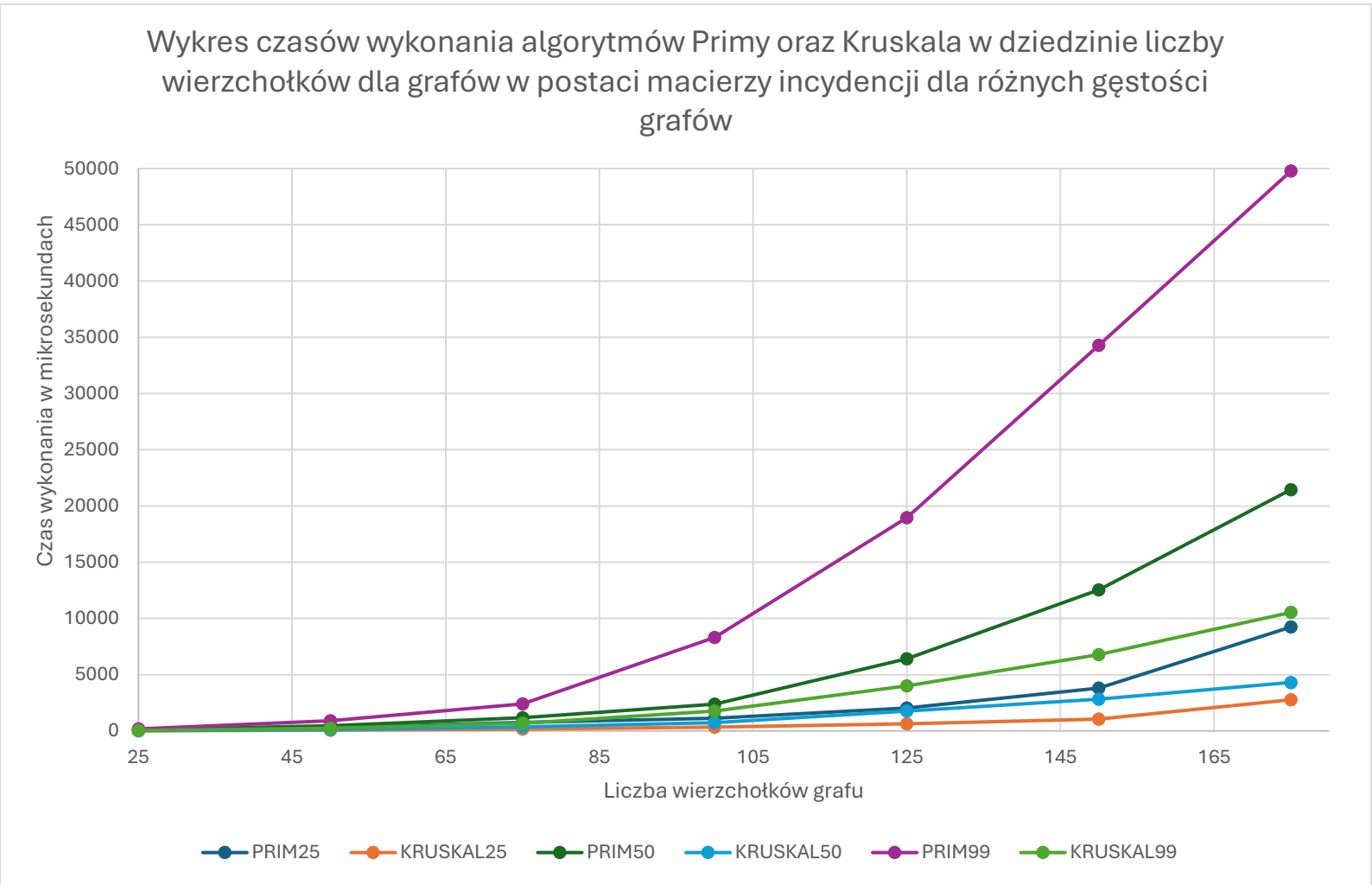


Tabela 3 Czasy działania algorytmów najkrótszej ścieżki w mikrosekundach dla grafu w postaci listy sąsiedztwa

Ilość wierzchołkow	DIJKSTRA25 [µs]	BELLMANFORD25 [µs]	DIJKSTRA50 [µs]	BELLMANFORD50 [µs]	DIJKSTRA99 [µs]	BELLMANFORD99 [µs]
25	4,578	3,052	6,394	4,588	8,412	7,344
50	13,746	10,066	16,42	15,37	21,3	28,652
75	21,902	23,554	36,252	44,308	51,954	63,888
100	34,142	34,026	62,508	62,324	185,292	227,948
125	64,592	54,358	159,37	156,208	308,036	425,644
150	102,05	85,2	218,51	299,562	407,446	694,566
175	179,305	228,695	286,51	420,895	539,36	871,28

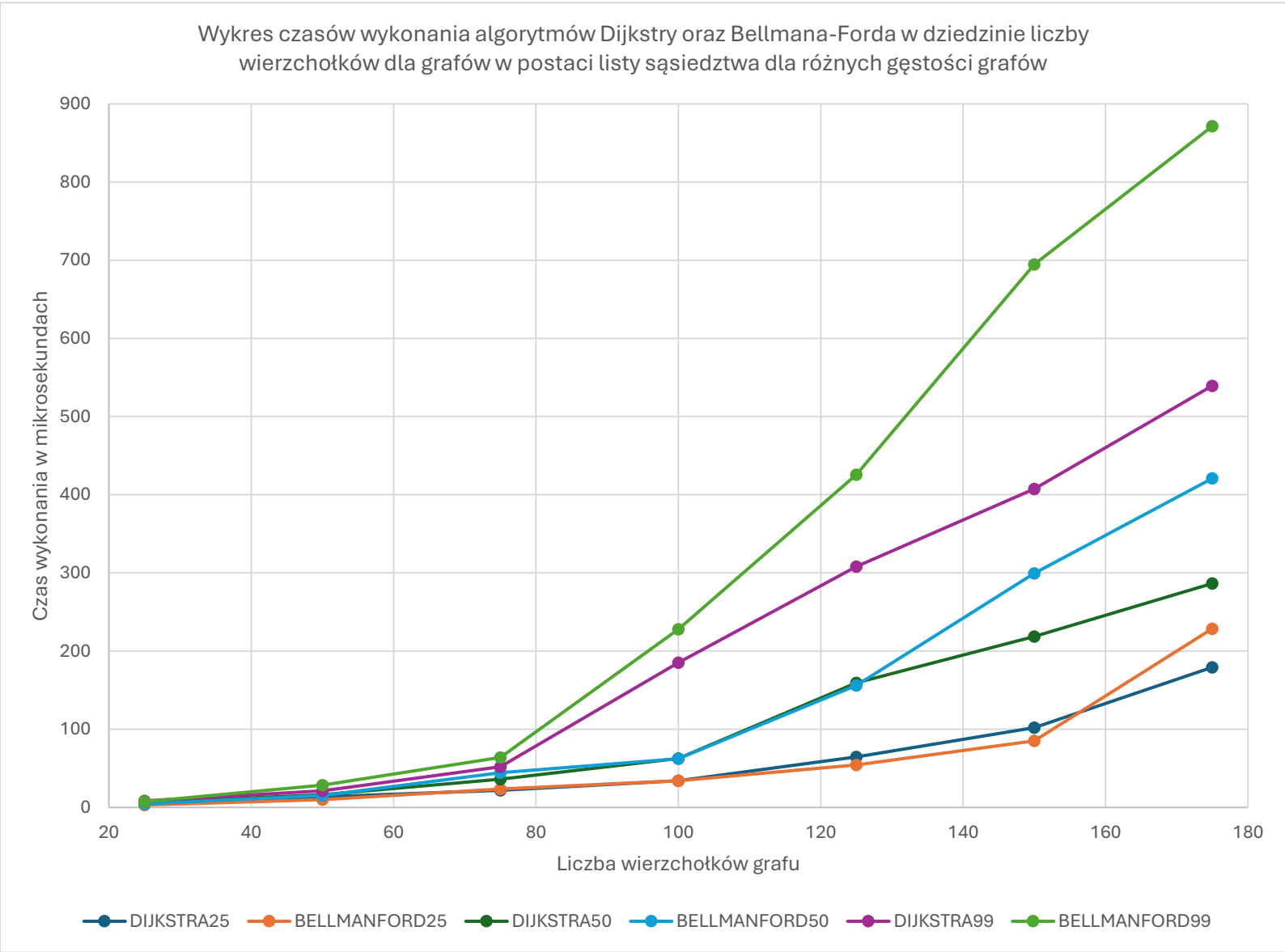
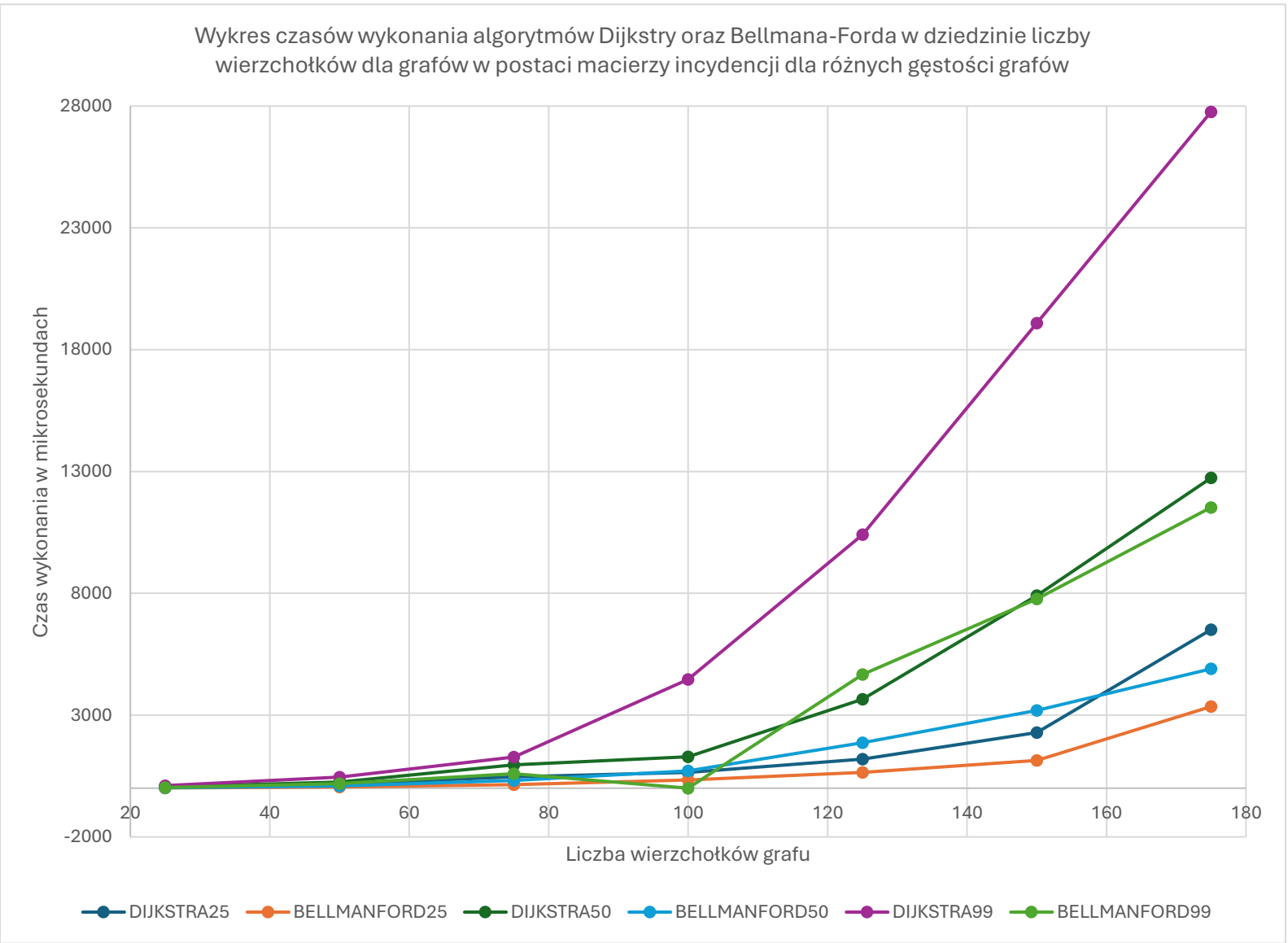


Tabela 4 Czasy działania algorytmów najkrótszej ścieżki w mikrosekundach dla grafu w postaci macierzy incydencji

Ilość wierzchołkow	DIJKSTRA25 [µs]	BELLMANFORD25 [µs]	DIJKSTRA50 [µs]	BELLMANFORD50 [µs]	DIJKSTRA99 [µs]	BELLMANFORD99 [µs]
25	21,77	9,68	56,73	16,898	106,61	32,384
50	161,86	49,476	252,63	95,996	457,042	188,472
75	463,948	142,866	960,246	318,432	1275,67	595,048
100	646,822	338,076	1292,45	707,478	4469,44	1767
125	1194,08	649,636	3648,84	1863,13	10413,5	4668,62
150	2282,16	1135,91	7908,94	3197,55	19093,5	7767,55
175	6509,5	3350,28	12736,3	4898,78	27766,8	11525,6



Kolejne tabele i wykresy przedstawiają podział wyników ze względu na gęstość grafu. Kolumny zostały opisane za pomocą symboli: nazwa algorytmu oraz reprezentacja grafu w pamięci komputera.

Tabela 5 Czasy działania algorytmów minimalnego drzewa rozpinającego w mikrosekundach dla grafu o gęstości 25%

Ilość wierzchołkow	PRIM_LIST [µs]	PRIM_MATRIX [µs]	KRUSKAL_LIST [µs]	KRUSKAL_MATRIX [µs]
25	6,726	49,122	7,866	11,594
50	20,648	293,902	29,448	61,488
75	36,06	757,46	68,928	163,006
100	54,256	1133,82	111,058	345,058
125	130,528	2035,72	196,03	645,126
150	241,994	3804,57	314,05	1043,02
175	365,575	9234,46	670,305	2768,99

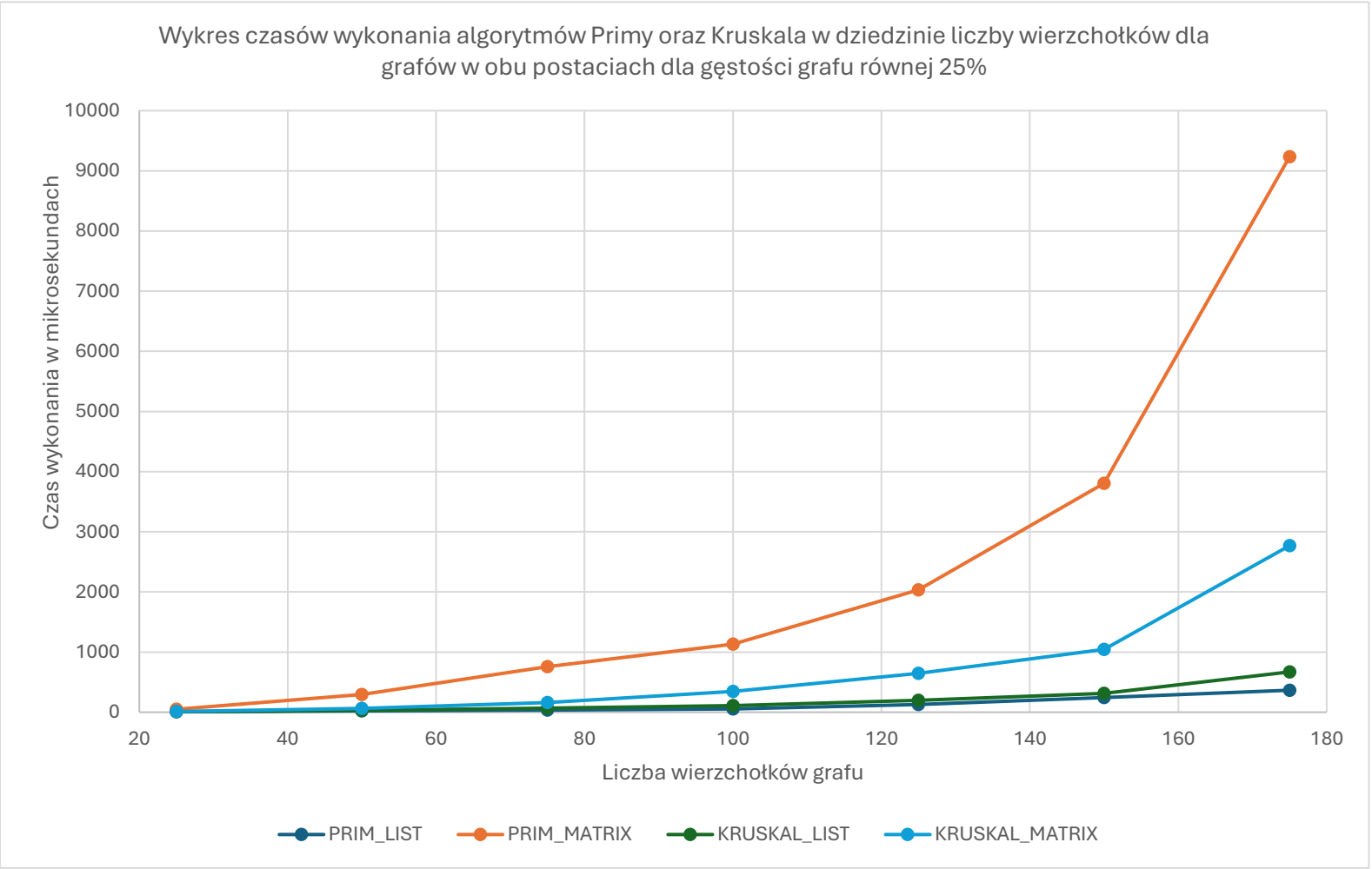


Tabela 6 Czasy działania algorytmów minimalnego drzewa rozpinającego w mikrosekundach dla grafu o gęstości 50%

Ilość wierzchołkow	PRIM_LIST [µs]	PRIM_MATRIX [µs]	KRUSKAL_LIST [µs]	KRUSKAL_MATRIX [µs]
25	8,844	102,112	13,754	22,652
50	25,512	450,54	56,454	118,954
75	56,32	1174,41	135,964	353,222
100	133,962	2382,89	240,468	721,334
125	355,948	6411,25	541,496	1769,9
150	528,518	12537,4	860,354	2821,52
175	700,535	21449,3	1160,28	4311,22

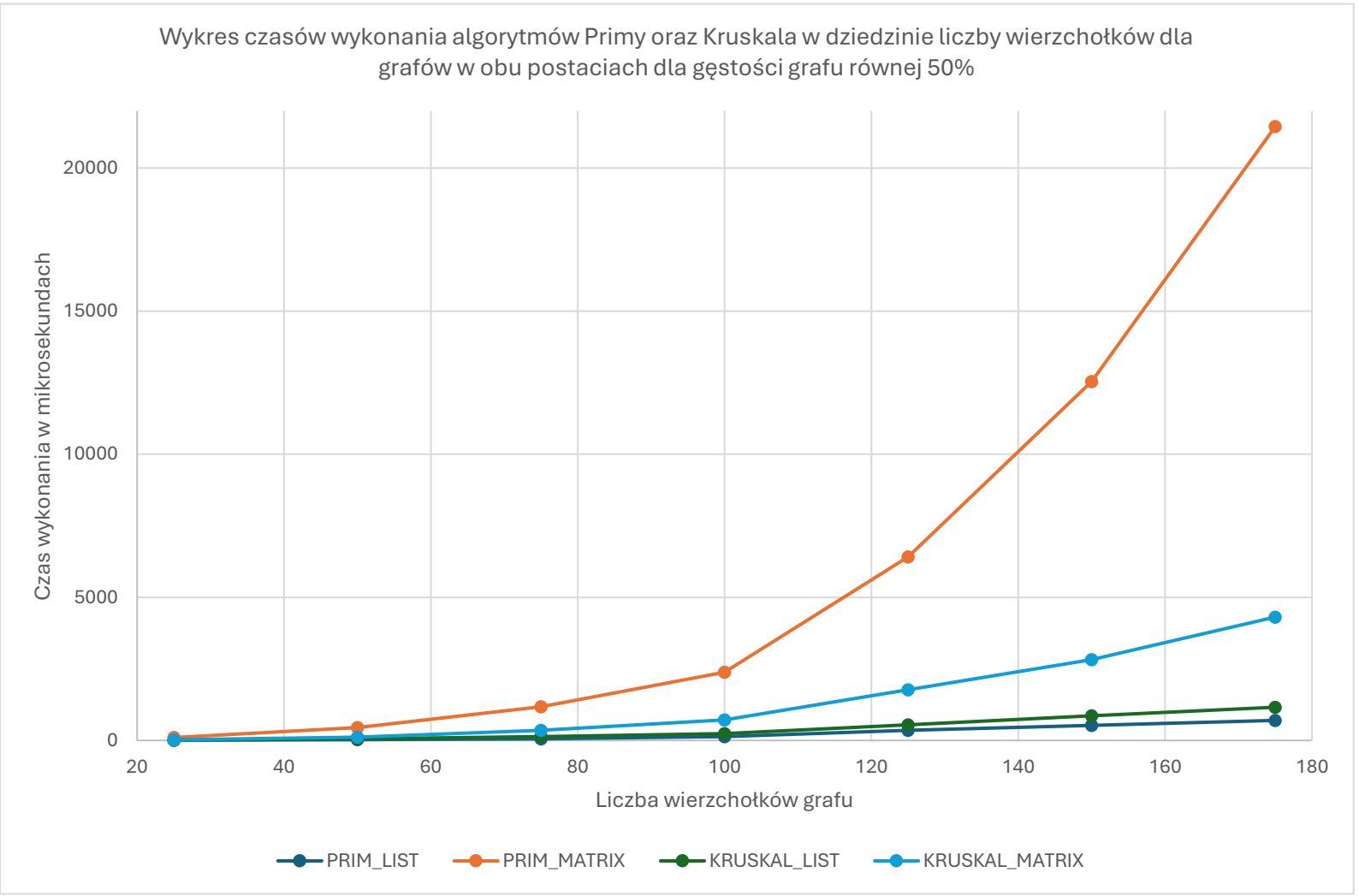


Tabela 7 Czasy działania algorytmów minimalnego drzewa rozpinającego w mikrosekundach dla grafu o gęstości 99%

Ilość wierzchołkow	PRIM_LIST [µs]	PRIM_MATRIX [µs]	KRUSKAL_LIST [µs]	KRUSKAL_MATRIX [µs]
25	12,436	181,194	25,236	42,488
50	42,006	914,084	117,54	244,156
75	98,76	2414,15	271,302	682,416
100	426,986	8301,46	729,262	1756,28
125	732,978	18970,8	1220,4	3999,83
150	1016,25	34271,5	2028,27	6782,34
175	1281,46	49768,4	2770,37	10534,7

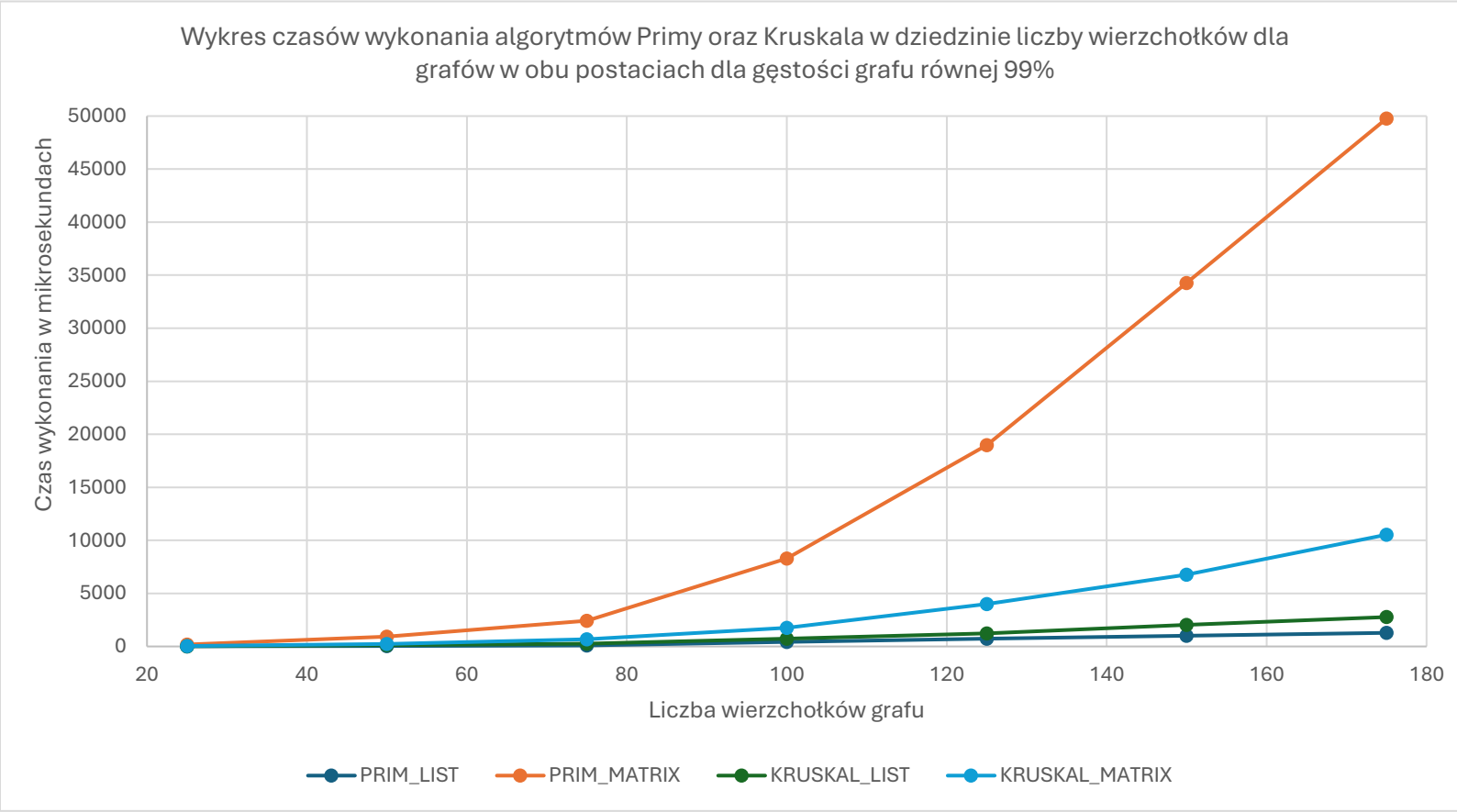


Tabela 8 Czasy działania algorytmów najkrótszej ścieżki w mikrosekundach dla grafu o gęstości 25%

Ilość wierzchołkow	DIJKSTRA_LIST [µs]	DIJKSTRA_MATRIX [µs]	BELLMANFORD_LIST [µs]	BELLMANFORD_MATRIX [µs]
25	4,578	21,77	3,052	9,68
50	13,746	161,86	10,066	49,476
75	21,902	463,948	23,554	142,866
100	34,142	646,822	34,026	338,076
125	64,592	1194,08	54,358	649,636
150	102,05	2282,16	85,2	1135,91
175	179,305	6509,5	228,695	3350,28

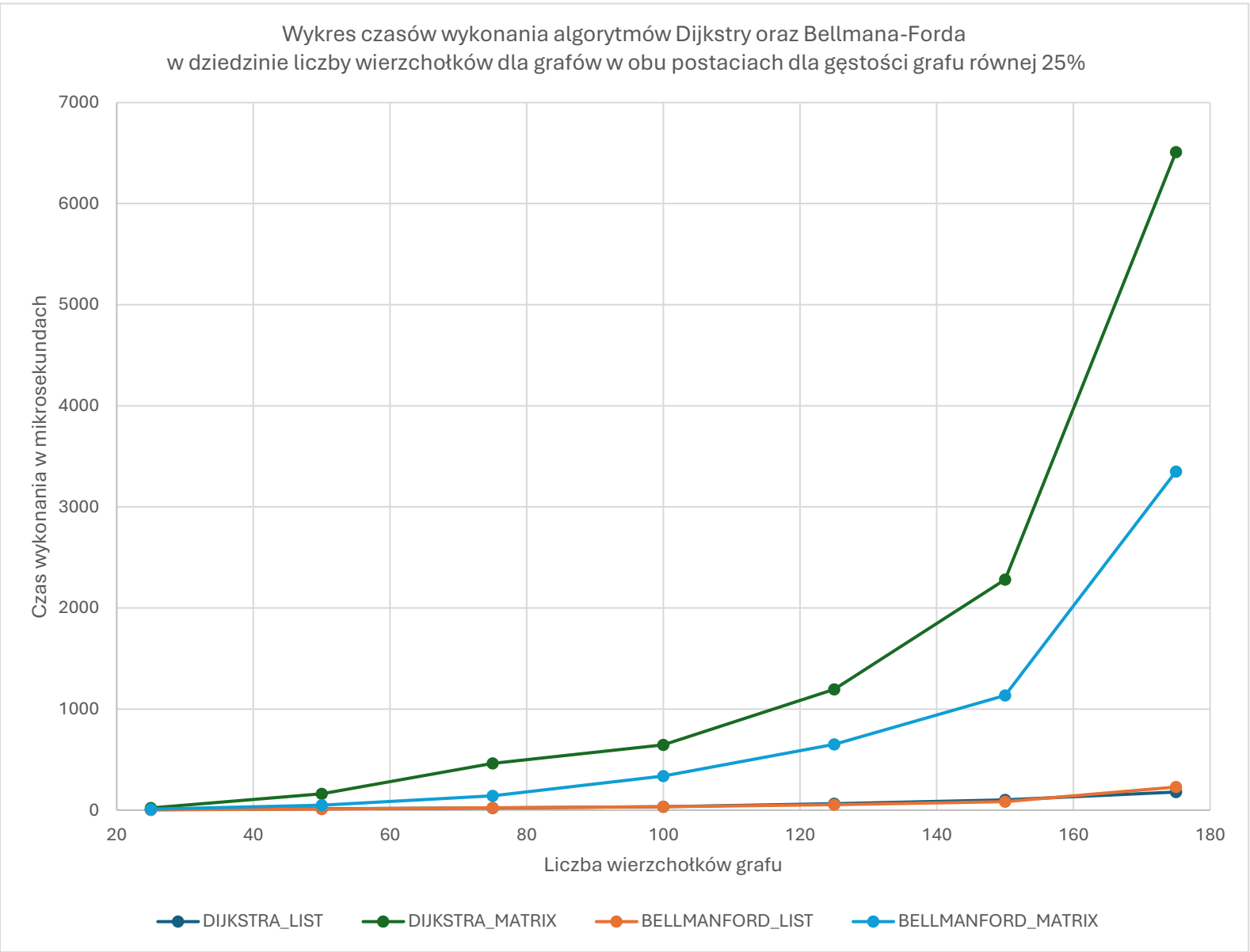


Tabela 9 Czasy działania algorytmów najkrótszej ścieżki w mikrosekundach dla grafu o gęstości 50%

Ilość wierzchołkow	DIJKSTRA_LIST [µs]	DIJKSTRA_MATRIX [µs]	BELLMANFORD_LIST [µs]	BELLMANFORD_MATRIX [µs]
25	6,394	56,73	4,588	16,898
50	16,42	252,63	15,37	95,996
75	36,252	960,246	44,308	318,432
100	62,508	1292,45	62,324	707,478
125	159,37	3648,84	156,208	1863,13
150	218,51	7908,94	299,562	3197,55
175	286,51	12736,3	420,895	4898,78

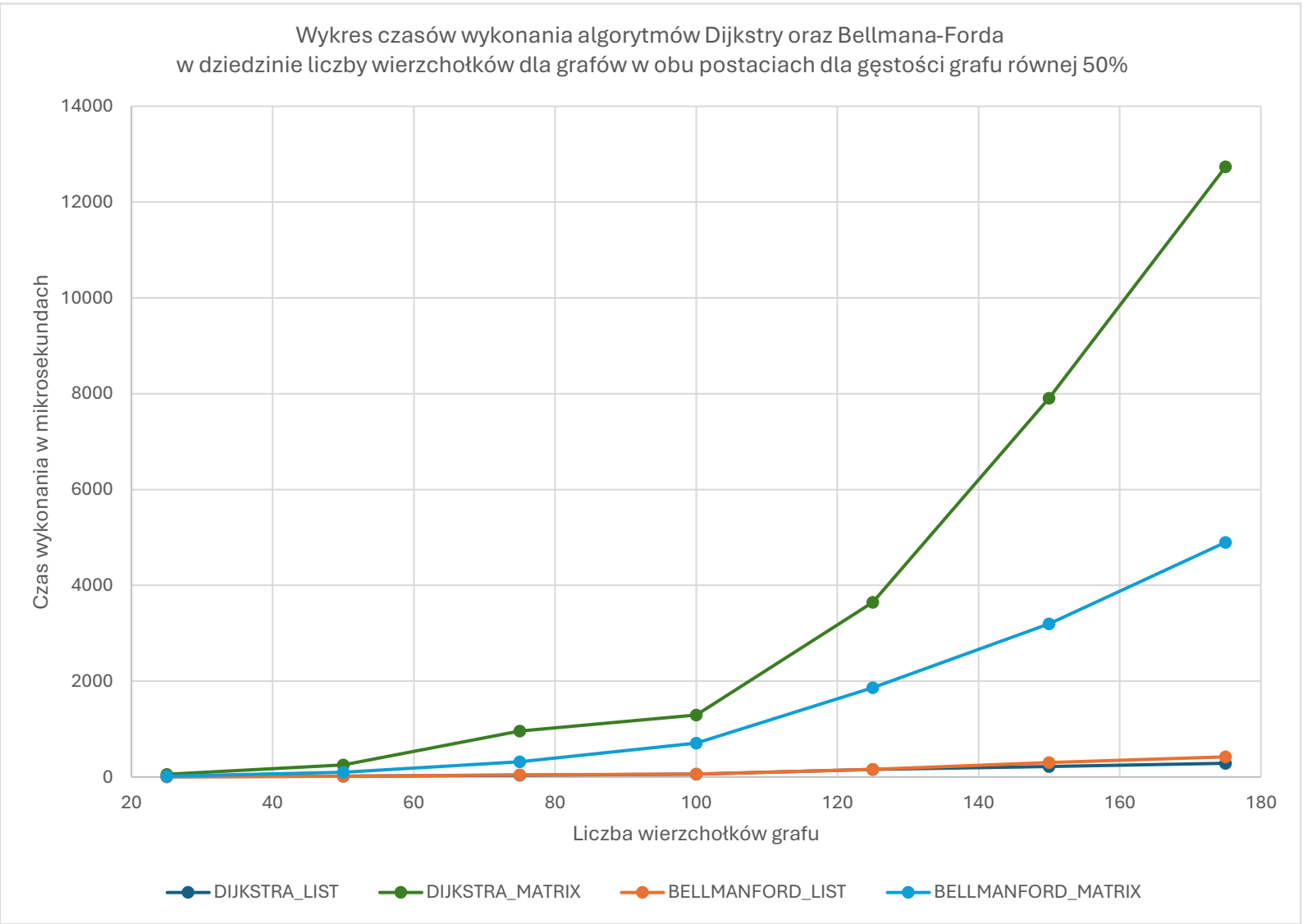
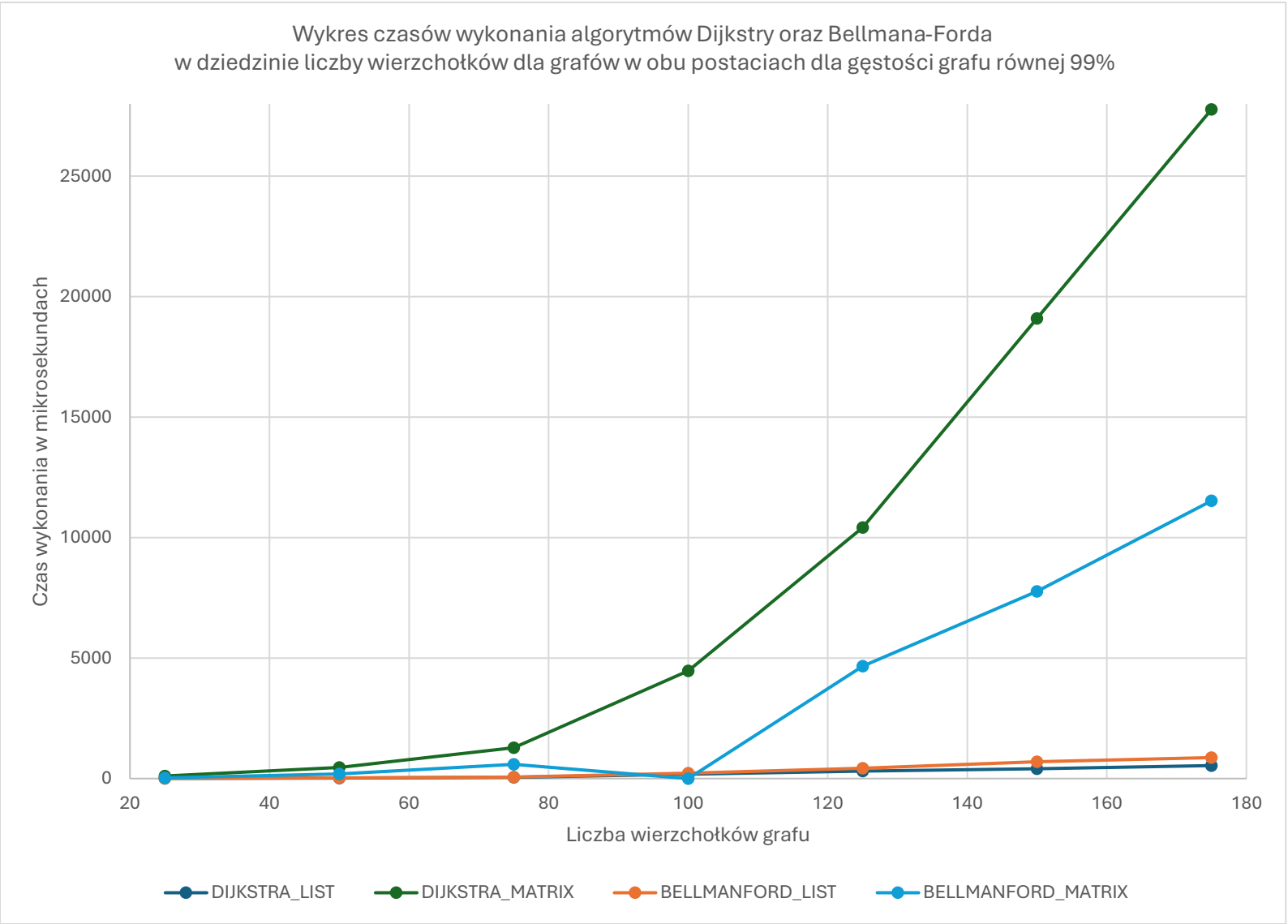


Tabela 10 Czasy działania algorytmów najkrótszej ścieżki w mikrosekundach dla grafu o gęstości 99%

Ilość wierzchołkow	DIJKSTRA_LIST [µs]	DIJKSTRA_MATRIX [µs]	BELLMANFORD_LIST [µs]	BELLMANFORD_MATRIX [µs]
25	8,412	106,61	7,344	32,384
50	21,3	457,042	28,652	188,472
75	51,954	1275,67	63,888	595,048
100	185,292	4469,44	227,948	1767
125	308,036	10413,5	425,644	4668,62
150	407,446	19093,5	694,566	7767,55
175	539,36	27766,8	871,28	11525,6



Wnioski

Na podstawie wyników dla algorytmów wyznaczających minimalne drzewo rozpinające można wysnuć następujące wnioski. Dla grafów w postaci listy sąsiedztwa czasy wykonania algorytmów są kilku a nawet kilkanaście razy szybsze niż dla grafów w postaci macierzy incydencji. Wraz ze wzrostem gęstości grafów, czasy trwania algorytmów rosną. W przypadku reprezentacji grafu za pomocą macierzy incydencji dla wszystkich gęstości grafu algorytm Prima wykonywał się średnio dłużej niż algorytm Kruskala. Dla listy sąsiedztwa algorytm Kruskala wykonywał się nieznacznie dłużej niż algorytm Prima.

W przypadku algorytmów do wyszukiwania najkrótszej ścieżki sytuacja prezentuje się następująco. Dla scenariusza gdzie graf przechowywany jest za pomocą macierzy incydencji algorytm Dijkstry uzyskuje większe czasy wykonania niż algorytm Bellmana-Forda. W przypadku reprezentacji grafu za pomocą listy sąsiedztwa czasy wykonania algorytmów są zbliżone.

Dla wszystkich przypadków charakterystyki poszczególnych algorytmów pokrywają się z teoretycznymi, jednakże nie widać tego na niektórych wykresach. Jest to spowodowane tym, że uzyskane wyniki między algorytmami różnią się czasem o rzędy wielkości co powoduje małą rozpiętość danych na wykresie dla pozostałych algorytmów.

Na podstawie wyżej uzyskanych wyników można stwierdzić, że lista sąsiedztwa jest bardziej wydajną strukturą danych w porównaniu do macierzy incydencji jeżeli mowa o używaniu algorytmów przebadanych w projekcie. Dla wszystkich scenariuszy uzyskała ona lepsze czasy o rzędy wielkości.