| INF PWR | **STRUKTURY DANYCH I ZŁOŻONOŚĆ OBLICZENIOWA**<br>Badanie efektywności algorytmów grafowych w zależności od rozmiaru instancji oraz sposobu reprezentacji grafu w pamięci komputera. | |
|---|---|---|
| Alicja Myśliwiec<br>Numer indeksu: 248867 | Wtorek 15.15 – 16.55<br>TP | Dr inż.<br>Dariusz Banasiak |

# 1. Cel ćwiczenia:

Celem projektu jest zaimplementowanie oraz dokonanie pomiaru czasu działania wybranych algorytmów grafowych rozwiązujących następujące problemy:

- Wyznaczanie minimalnego drzewa rozpinającego (MST) - algorytm Prima oraz algorytm Kruskala
- Wyznaczanie najkrótszej ścieżki w grafie - algorytm Dijkstry oraz algorytm Bellmana - Forda

Algorytmy te należało zaimplementować dla obu poniższych reprezentacji grafu w pamięci komputera:

- reprezentacja macierzowa (macierz incydencji)
- reprezentacja listowa (lista następników/poprzedników)

# 2. Wstęp teoretyczny

Reprezentacja grafu to sposób zapisu grafu umożliwiający jego obróbkę z użyciem programów komputerowych.

## Lista sąsiedztwa

Reprezentacja grafów przez listy sąsiedztwa, jak sama nazwa wskazuje, polega na trzymaniu dla każdego wierzchołka listy jego wszystkich sąsiadów (następników albo poprzedników) oraz ich wag krawędzi.

- złożoność pamięciowa: $O(E)$
- przejrzenie wszystkich krawędzi: $O(E)$
- przejrzenie następników/poprzedników danego wierzchołka: maksymalnie $O(V)$ (tyle sąsiadów może mieć wierzchołek), ale średnio $O(E/V)$ (trzeba przejrzeć całą listę sąsiadów, których średnio wierzchołek ma właśnie $E/V$)
- sprawdzenie istnienia jednej krawędzi: tak jak przedstawiono wyżej w przypadku przeglądania następników/poprzedników (trzeba przejrzeć listę dla pewnego wierzchołka)

Jest to jedna z lepszych reprezentacji - minimalna możliwa złożoność pamięciowa, szybkie przeszukiwanie krawędzi wychodzących z danego wierzchołka, stosunkowo łatwa implementacja. Wadą jest jedynie długi

czas sprawdzenia istnienia pojedynczej krawędzi. Można go poprawić do $O(log\ V)$ wprowadzając drzewa BST (albo nawet drzewa zrównoważone) zamiast list, ale to skomplikuje implementację.

## Macierz incydencji

Macierz incydencji składa się z V wierszy (odpowiadającym wierzchołkom) i E kolumn (odpowiadającym krawędziom). Na „skrzyżowaniu" wierzchołka z krawędzią jest -1 gdy krawędź wychodzi z wierzchołka, +1 - krawędź wchodzi, 2 - pętla, 0 - brak incydencji. W przypadku, gdy uwzględniamy wagi krawędzi, na „skrzyżowaniu" wierzchołka z krawędzią jest uwzględniona waga tej krawędzi, która wchodzi lub wychodzi z wierzchołka.

- złożoność pamięciowa: $O(V * E)$
- przejrzenie wszystkich krawędzi: $O(E)$
- przejrzenie następników/poprzedników danego wierzchołka: $O(E)$
- sprawdzenie istnienia jednej krawędzi: $O(E)$

Reprezentacja nie wygląda na dobrą ze względu na dużą złożoność pamięciową.

## Minimalne drzewo rozpinające

Drzewo rozpinające to takie, które zawiera wszystkie wierzchołki grafu oraz niektóre z jego krawędzi. Należy pamiętać, że drzewa nie zawierają cykli. Natomiast minimalne drzewo rozpinające jest drzewem rozpinającym, którego suma wag krawędzi jest najmniejsza ze wszystkich pozostałych drzew rozpinających danego grafu. W grafie może istnieć kilka drzew o tych własnościach.

## Algorytm Prima

Jest to algorytm zachłanny wyznaczający minimalne drzewo rozpinające. Algorytm został wynaleziony w 1930 przez czeskiego matematyka Vojtěcha Jarníka, a następnie odkryty na nowo przez informatyka Roberta C. Prima w 1957 oraz niezależnie przez Edsgera Dijkstrę w 1959. Z tego powodu algorytm nazywany jest również czasami algorytmem Dijkstry-Prima, algorytmem DJP, algorytmem Jarníka, albo algorytmem Prima-Jarníka. Algorytm wymaga wyszukiwania krawędzi o najniższym koszcie. Do tego celu można wykorzystać kolejkę priorytetową, w której elementy są udostępniane według najniższych wag. W momencie dodania wierzchołka do drzewa, w kolejce umieszczamy wszystkie krawędzie prowadzące od tego wierzchołka do wierzchołków, które jeszcze nie znajdują się w drzewie rozpinającym. Następnie z kolejki pobieramy krawędzie dotąd, aż otrzymamy krawędź prowadzącą do jeszcze niewybranego wierzchołka. Krawędź tą dodajemy do drzewa rozpinającego. Algorytm musi sprawdzać, czy krawędź

pobrana z kolejki priorytetowej łączy się z niewybranym jeszcze wierzchołkiem. Do tego celu można wykorzystać prostą tablicę logiczną odwiedzin *visited*. Tablica ta odwzorowuje wierzchołki grafu. Początkowo wszystkie jej elementy ustawiamy na false, co oznacza, że odpowiadające im wierzchołki nie znajdują się w drzewie rozpinającym. Następnie w miarę dodawania krawędzi do drzewa objęte nimi wierzchołki oznaczamy w tablicy *visited* wartością true. Wybieramy w grafie dowolny wierzchołek startowy. Dopóki drzewo nie pokrywa całego grafu, znajdujemy krawędź o najniższym koszcie spośród wszystkich krawędzi prowadzących od wybranych już wierzchołków do wierzchołków jeszcze niewybranych. Znalezioną krawędź dodajemy do drzewa rozpinającego. Złożoność obliczeniowa zależy od implementacji kolejki priorytetowej:

- Dla wersji opartej na zwykłym kopcu (bądź drzewie czerwono-czarnym): $O(|E| * log|V|)$
- Przy zastosowaniu kopca Fibonacciego $O(|E| + |V| * log|V|)$

## Algorytm Kruskala

Jest to algorytm grafowy wyznaczający minimalne drzewo rozpinające dla grafu nieskierowanego ważonego, o ile jest on spójny. Jest to przykład algorytmu zachłannego. Został on po raz pierwszy opublikowany przez Josepha Kruskala w 1956 roku w czasopiśmie Proceedings of the American Mathematical Society. Aby zrealizować praktycznie ten algorytm, musimy dobrać odpowiednie struktury danych do wykonywanych w algorytmie operacji. Do drzewa dodajemy krawędzie. Lista musi być posortowana. Można tutaj wykorzystać zwykłą listę, dodać do niej wszystkie krawędzie grafu i posortować ją rosnąco względem ich wag. Lepszym rozwiązaniem może okazać się odpowiednio skonstruowana kolejka priorytetowa, która będzie przechowywać krawędzie w kolejności od najmniejszej do największej wagi. Można tutaj wykorzystać kolejkę opartą na kopcu. Algorytm wymaga sprawdzania, czy dodawana do drzewa krawędź nie tworzy cyklu z krawędziami, które już znajdują się w drzewie. Wykrywanie cyklu jest czasochłonne. Lepszym rozwiązaniem jest zastosowanie struktury zbiorów rozłącznych, w której przechowywane są wierzchołki grafu. Przed dodaniem krawędzi u–v do drzewa sprawdzamy, czy wierzchołki u i v znajdują się w rozłącznych zbiorach. Jeśli tak, to zbiory te łączymy i krawędź dodajemy do drzewa. Jeśli nie, to oznacza to, iż u oraz v należą do tej samej spójnej składowej i dodanie krawędzi spowodowałoby powstanie cyklu. W takim przypadku krawędź odrzucamy. Jako zbiór można wziąć tablicę wszystkich krawędzi posortowaną według wag. Wtedy w każdym kroku najmniejsza krawędź to po prostu następna w kolejności. Sortowanie działa w czasie $O(ElogE) = O(ElogV)$. Drugą fazę algorytmu można zrealizować przy pomocy struktury zbiorów rozłącznych – na początku każdy wierzchołek tworzy osobny zbiór, struktura pozwala na sprawdzenie, czy dwa wierzchołki są w jednym zbiorze i ewentualne połączenie dwu zbiorów w jeden. Przy implementacji przez tzw. las drzew rozłącznych z kompresją ścieżki operacje te łącznie działają w czasie $O\big(E\alpha(E,V)\big)$, gdzie α jest jest niezwykle wolno rosnącą funkcją (odwrotnością funkcji Ackermanna).

## Najkrótsza ścieżka w grafie

Problem najkrótszej ścieżki to zagadnienie polegające na znalezieniu w grafie ważonym najkrótszego połączenia pomiędzy danymi wierzchołkami. Szczególnymi przypadkami tego problemu są problem najkrótszej ścieżki od jednego wierzchołka do wszystkich innych oraz problem najkrótszej ścieżki pomiędzy wszystkimi parami wierzchołków. Okazuje się, że żeby znaleźć najkrótszą ścieżkę pomiędzy dwoma wierzchołkami grafu trzeba (w pesymistycznym przypadku) znaleźć najkrótsze ścieżki od wierzchołka wyjściowego do wszystkich innych wierzchołków. Problem najkrótszej ścieżki od jednego z wierzchołków do wszystkich innych można więc zobrazować jako problem znalezienia najkrótszej drogi pomiędzy dwoma miastami. W takim wypadku wierzchołkami grafu są skrzyżowania dróg, krawędziami – drogi, a wagi krawędzi odwzorowują długość danego odcinka drogowego. Drugi szczególny przypadek problemu najkrótszej ścieżki występuje, gdy chcemy znaleźć najkrótsze ścieżki pomiędzy każdą parą wierzchołków. Możliwe jest zrobienie tego dla każdego wierzchołka, używając algorytmu znajdującego najkrótszą ścieżkę od jednego wierzchołka do wszystkich innych, jednak metoda ta okazuje się w praktyce niezbyt efektywna.

## Algorytm Dijkstry

Jest to algorytm opracowany przez holenderskiego informatyka Edsgera Dijkstrę. Służy do znajdowania najkrótszej ścieżki z pojedynczego źródła w grafie o nieujemnych wagach krawędzi. 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, jest przykładem algorytmu zachłannego. Z algorytmu Dijkstry można skorzystać przy obliczaniu najkrótszej drogi do danej miejscowości. Wystarczy przyjąć, że każdy z punktów skrzyżowań dróg to jeden z wierzchołków grafu, a odległości między punktami to wagi krawędzi. Jest często używany w sieciach komputerowych, np. przy trasowaniu (przykładowo w protokole OSPF). Złożoność obliczeniowa algorytmu Dijkstry zależy od liczby V wierzchołków i E krawędzi grafu. O rzędzie złożoności decyduje implementacja kolejki priorytetowej:

- Wykorzystując „naiwną” implementację poprzez zwykłą tablicę, otrzymujemy algorytm o złożoności $O(V^2)$
- W implementacji kolejki poprzez kopiec, złożoność wynosi $O(ElogV)$
- Po zastąpieniu zwykłego kopca kopcem Fibonacciego złożoność zmienia się na $O(E + VlogV)$

Pierwszy wariant jest optymalny dla grafów gęstych, drugi jest szybszy dla grafów rzadkich, trzeci jest bardzo rzadko używany ze względu na duży stopień skomplikowania i niewielki w porównaniu z nim zysk czasowy.

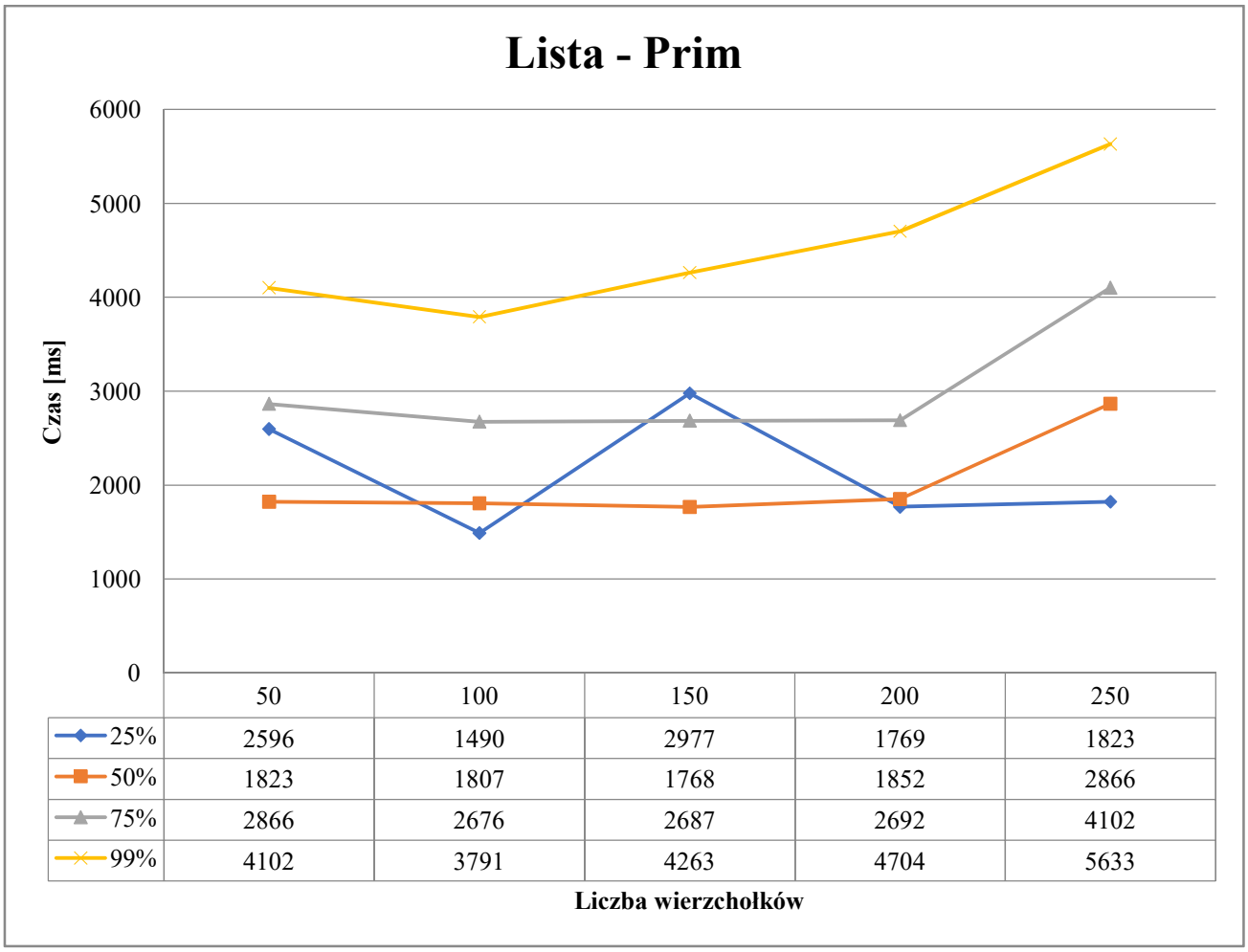
### Algorytm Bellmana - Forda

Jes to algorytm służący do wyszukiwania najkrótszych ścieżek w grafie ważonym z wierzchołka źródłowego do wszystkich pozostałych wierzchołków. Idea 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 $O(|V| * |E|)$. Złożoność pamięciowa wynosi $O(|V|)$. Algorytm może być wykorzystywany także do sprawdzania, czy w grafie występują ujemne cykle osiągalne ze źródła. Na algorytmie Bellmana-Forda bazuje protokół RIP - Routing Information Protocol.

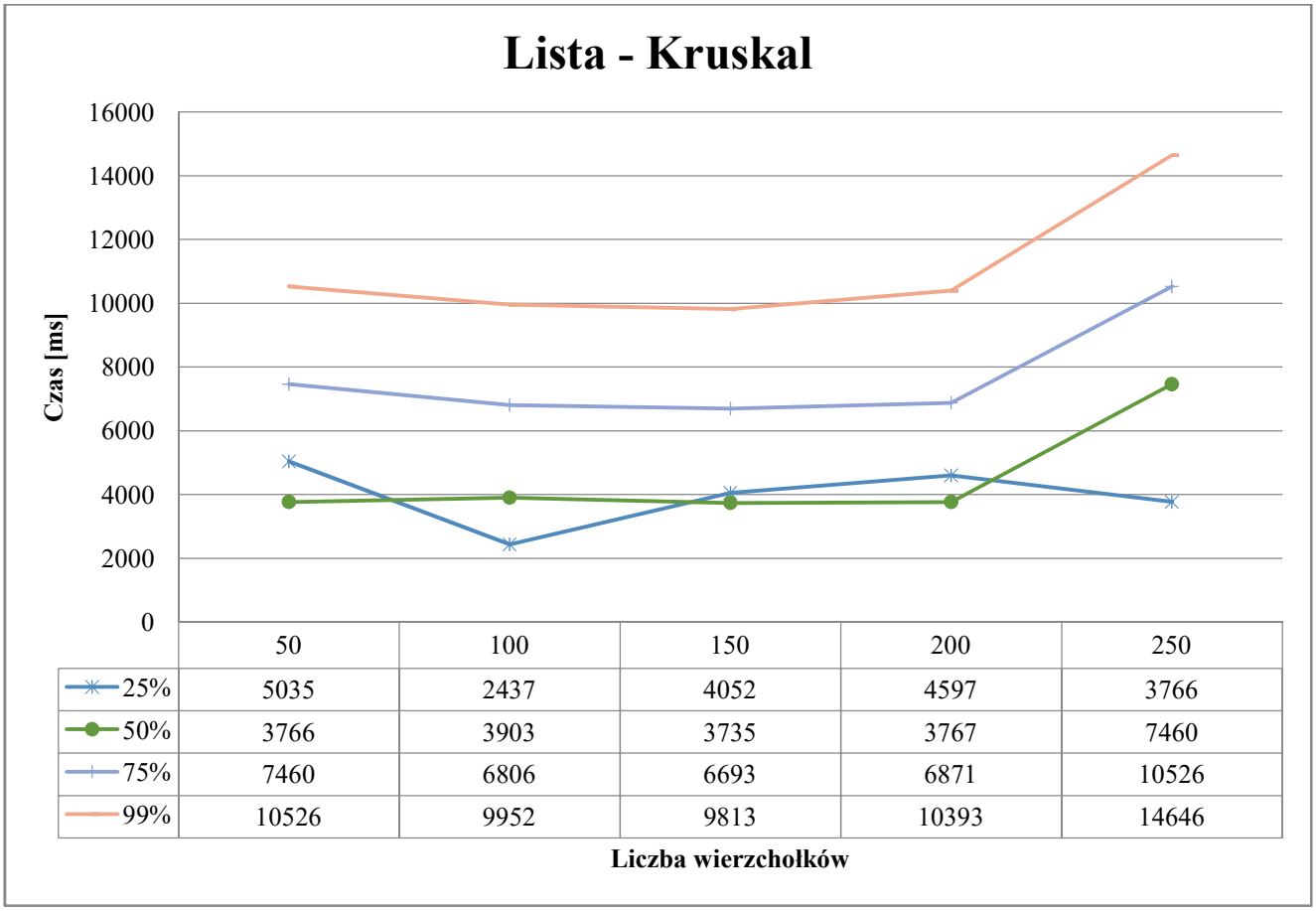
## 3. Plan projektu

Zadanie polegało na pomiarze czasu działania poszczególnych algorytmów w zależności od rozmiaru grafu i jego gęstości. Badania zostały wykonane dla 5 różnych liczb wierzchołków: 50, 100, 150, 200, 250 oraz następujących gęstości grafu: 25%, 50%, 75%, 99%. Dla każdego zestawu została wygenerowana seria losowych instancji. W sprawozdaniu zostały umieszczone uśrednione wyniki z danej serii. Wszystkie struktury są alokowane dynamiczne, a przepustowość krawędzi jest liczbą całkowitą. Program umożliwia również sprawdzenie poprawności zaimplementowanych operacji. Czas mierzony jest poprzez pobranie czasu systemowego przed i po wykonaniu algorytmu, a następnie obliczeniu różnicy.

## 4. Wyniki

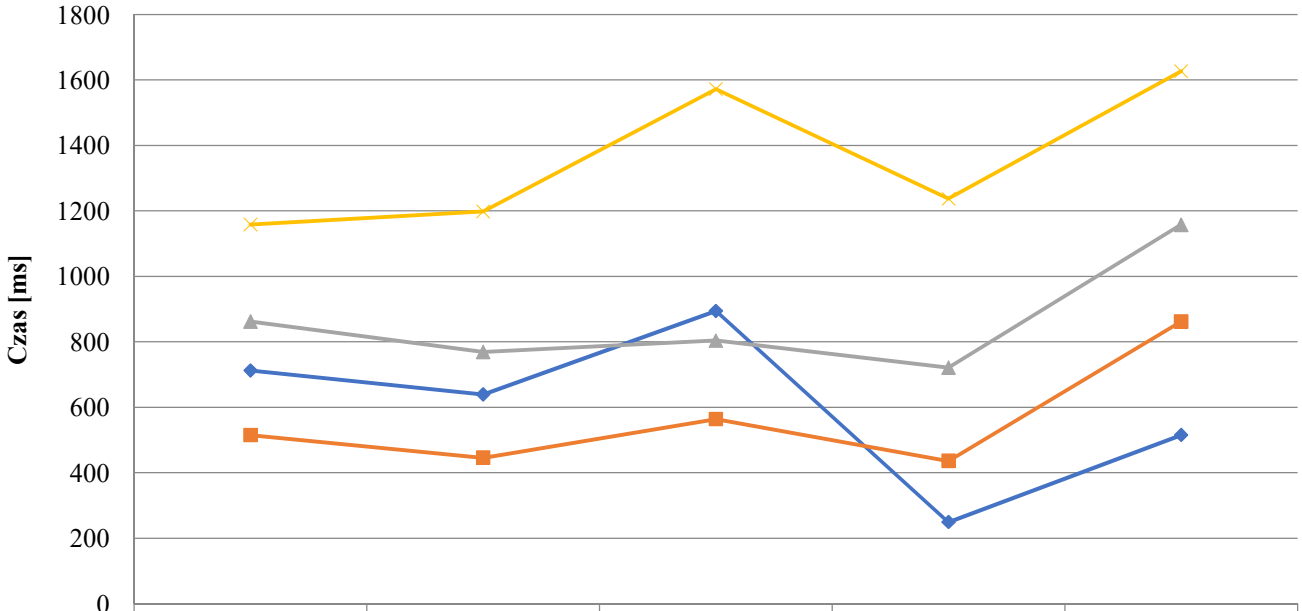



| | 50 | 100 | 150 | 200 | 250 |
|---|---|---|---|---|---|
| 25% | 2596 | 1490 | 2977 | 1769 | 1823 |
| 50% | 1823 | 1807 | 1768 | 1852 | 2866 |
| 75% | 2866 | 2676 | 2687 | 2692 | 4102 |
| 99% | 4102 | 3791 | 4263 | 4704 | 5633 |




| | 50 | 100 | 150 | 200 | 250 |
|---|---|---|---|---|---|
| 25% | 5035 | 2437 | 4052 | 4597 | 3766 |
| 50% | 3766 | 3903 | 3735 | 3767 | 7460 |
| 75% | 7460 | 6806 | 6693 | 6871 | 10526 |
| 99% | 10526 | 9952 | 9813 | 10393 | 14646 |

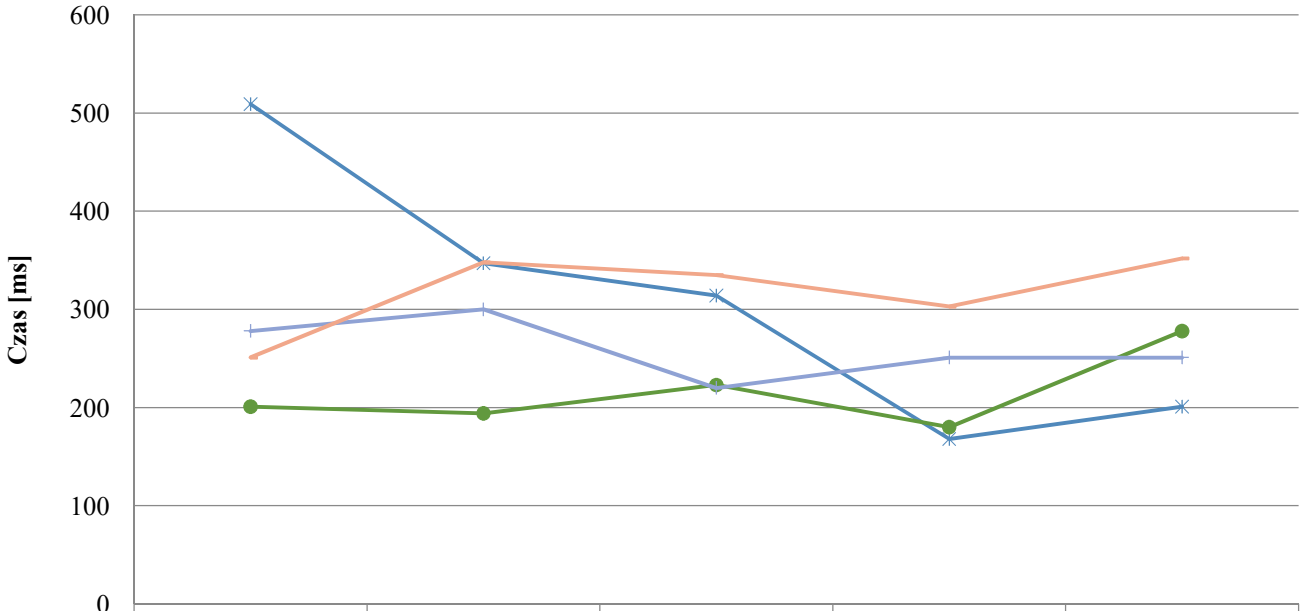
# Lista - Dijkstry




| | 50 | 100 | 150 | 200 | 250 |
|---|---|---|---|---|---|
| 25% | 712 | 639 | 894 | 249 | 515 |
| 50% | 515 | 446 | 564 | 436 | 862 |
| 75% | 862 | 769 | 804 | 721 | 1158 |
| 99% | 1158 | 1198 | 1572 | 1237 | 1627 |

Liczba wierzchołków

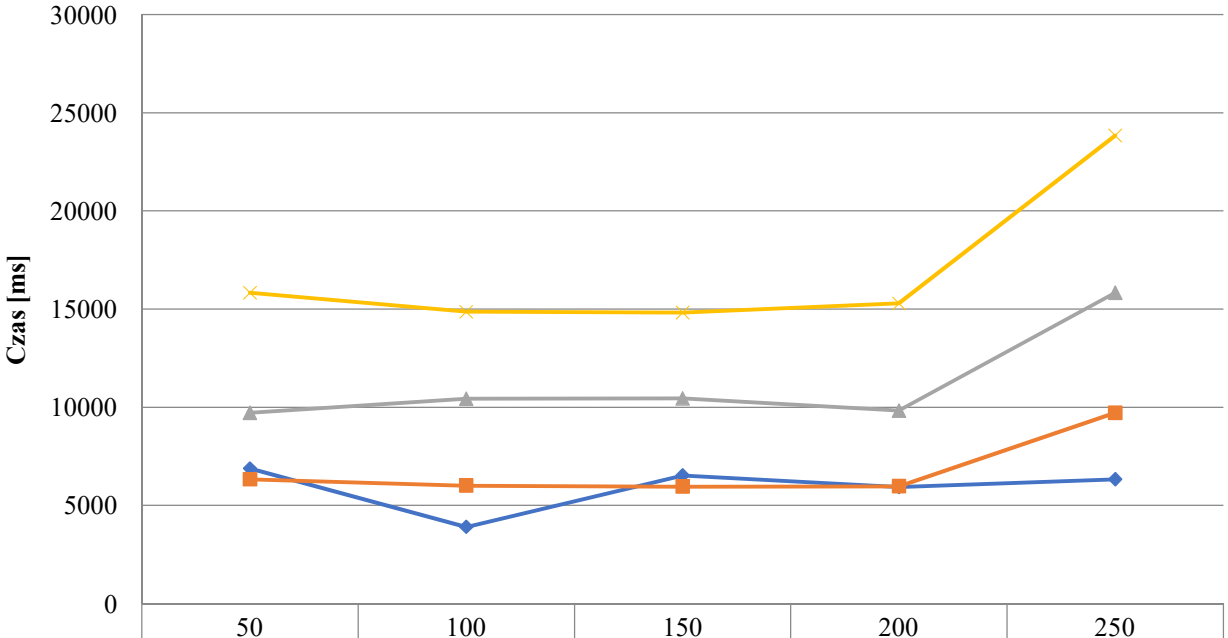
# Lista - Bellman-Ford




| | 50 | 100 | 150 | 200 | 250 |
|---|---|---|---|---|---|
| 25% | 509 | 347 | 314 | 168 | 201 |
| 50% | 201 | 194 | 223 | 180 | 278 |
| 75% | 278 | 300 | 220 | 251 | 251 |
| 99% | 251 | 348 | 335 | 303 | 352 |

Liczba wierzchołków

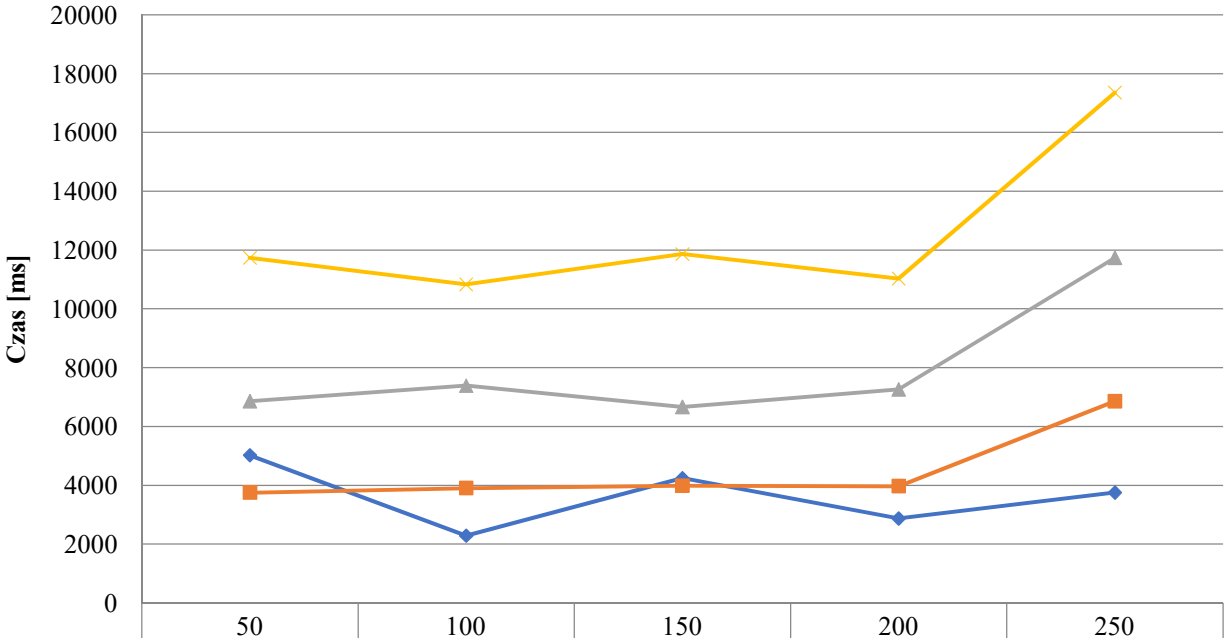
# Macierz - Prim




| | 50 | 100 | 150 | 200 | 250 |
|---|---|---|---|---|---|
| 25% | 6884 | 3908 | 6529 | 5937 | 6332 |
| 50% | 6332 | 6010 | 5960 | 5976 | 9717 |
| 75% | 9717 | 10438 | 10459 | 9832 | 15840 |
| 99% | 15840 | 14867 | 14826 | 15292 | 23843 |

Liczba wierzchołków

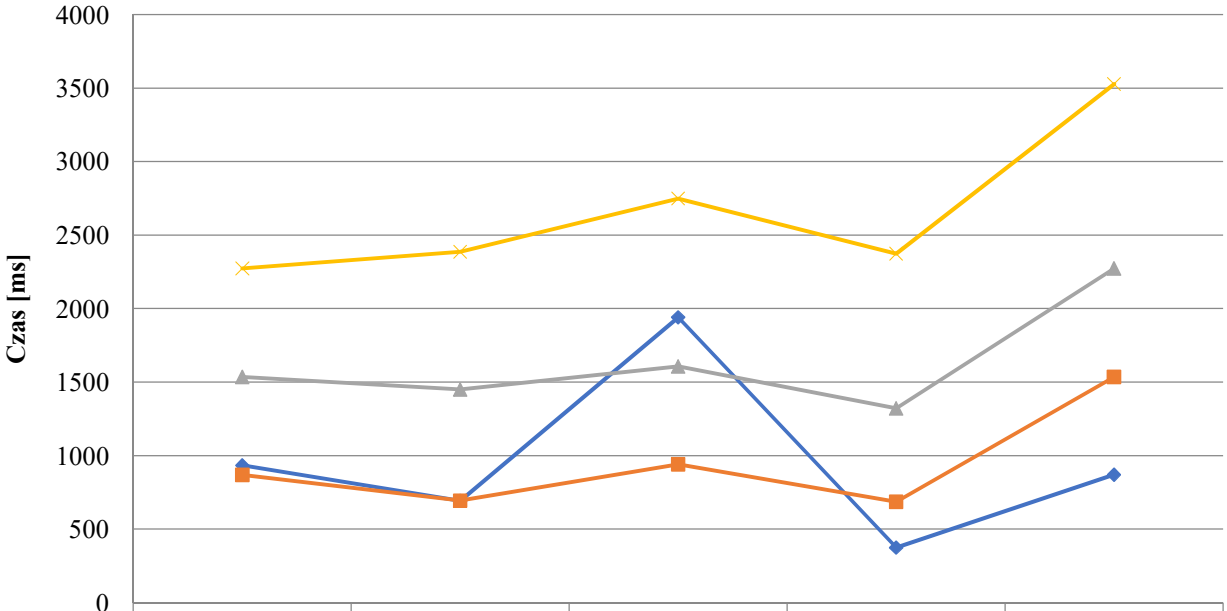
# Macierz - Kruskal




| | 50 | 100 | 150 | 200 | 250 |
|---|---|---|---|---|---|
| 25% | 5023 | 2284 | 4243 | 2870 | 3753 |
| 50% | 3753 | 3907 | 3991 | 3971 | 6862 |
| 75% | 6862 | 7388 | 6664 | 7264 | 11736 |
| 99% | 11736 | 10832 | 11863 | 11033 | 17361 |

Liczba wierzchołków

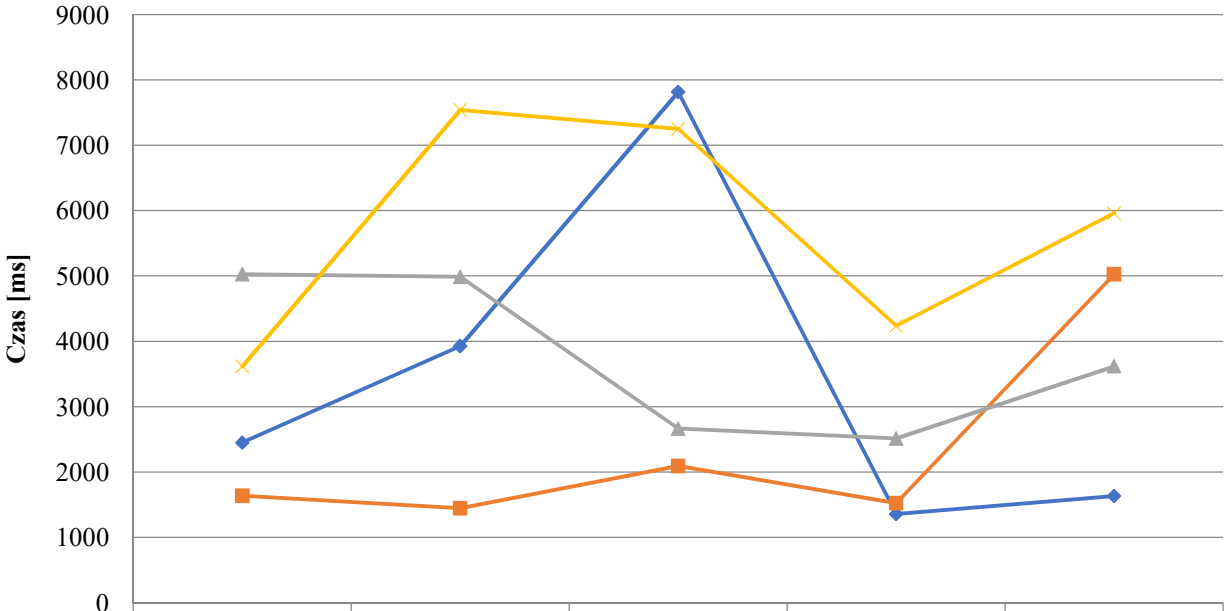
## Macierz - Dijkstry




| | 50 | 100 | 150 | 200 | 250 |
|---|---|---|---|---|---|
| 25% | 934 | 692 | 1941 | 374 | 870 |
| 50% | 870 | 695 | 942 | 687 | 1536 |
| 75% | 1536 | 1451 | 1607 | 1322 | 2273 |
| 99% | 2273 | 2387 | 2749 | 2374 | 3528 |

Liczba wierzchołków

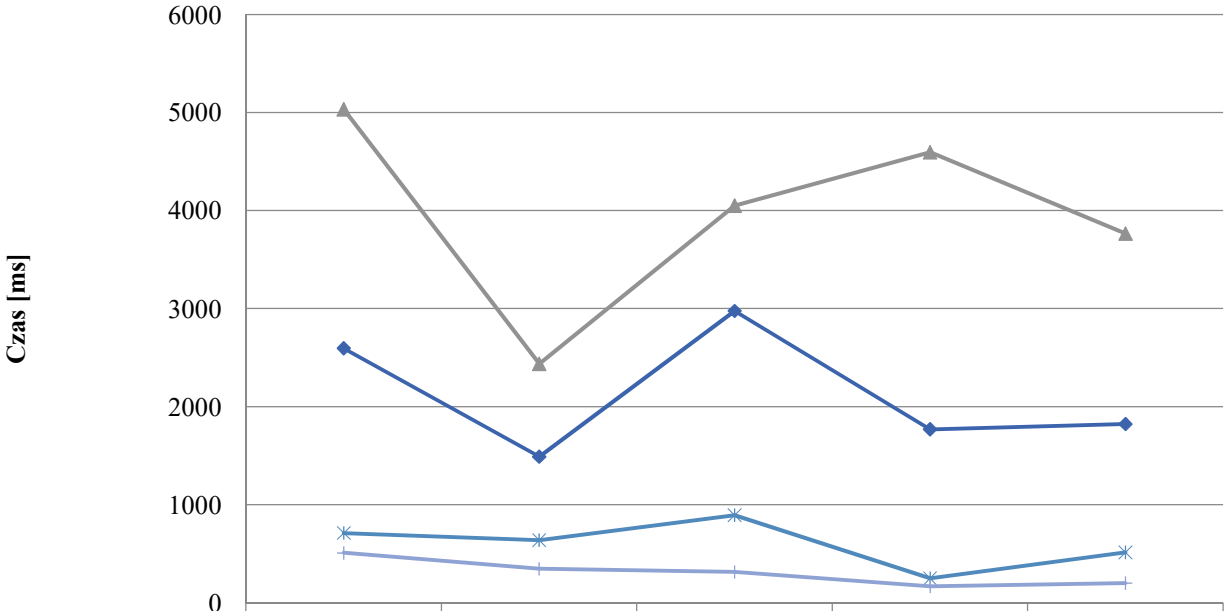
## Macierz - Bellman-Ford




| | 50 | 100 | 150 | 200 | 250 |
|---|---|---|---|---|---|
| 25% | 2451 | 3928 | 7816 | 1359 | 1635 |
| 50% | 1635 | 1444 | 2092 | 1523 | 5027 |
| 75% | 5027 | 4987 | 2666 | 2513 | 3618 |
| 99% | 3618 | 7541 | 7251 | 4239 | 5962 |

Liczba wierzchołków

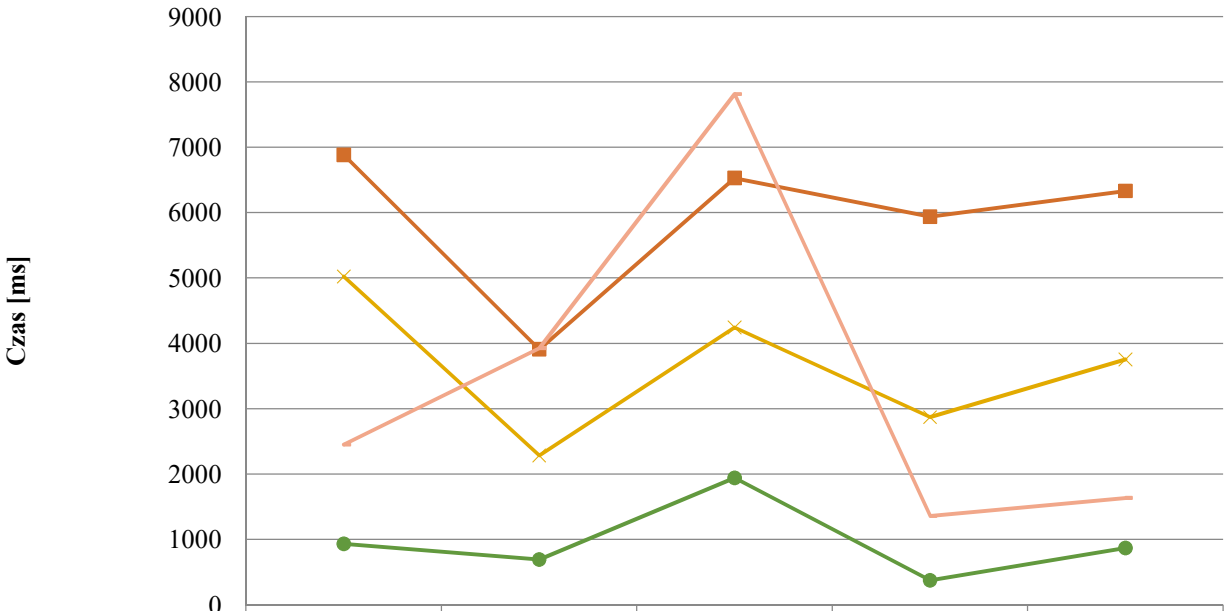
## Gęstość 25% - Lista




| | 50 | 100 | 150 | 200 | 250 |
|---|---|---|---|---|---|
| Prim | 2596 | 1490 | 2977 | 1769 | 1823 |
| Kruskal | 5035 | 2437 | 4052 | 4597 | 3766 |
| Dijkstry | 712 | 639 | 894 | 249 | 515 |
| Bellaman-Ford | 509 | 347 | 314 | 168 | 201 |

**Liczba wierzchołków**

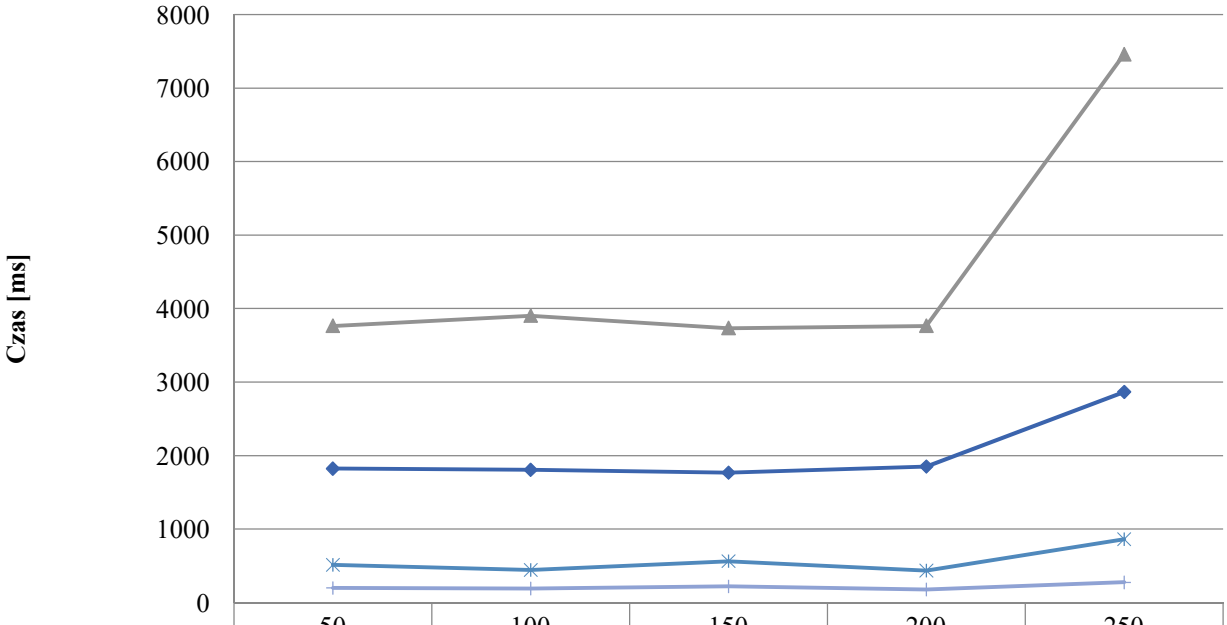
## Gęstość 25% - Macierz




| | 50 | 100 | 150 | 200 | 250 |
|---|---|---|---|---|---|
| Prim | 6884 | 3908 | 6529 | 5937 | 6332 |
| Kruskal | 5023 | 2284 | 4243 | 2870 | 3753 |
| Dijkstry | 934 | 692 | 1941 | 374 | 870 |
| Bellaman-Ford | 2451 | 3928 | 7816 | 1359 | 1635 |

**Liczba wierzchołków**

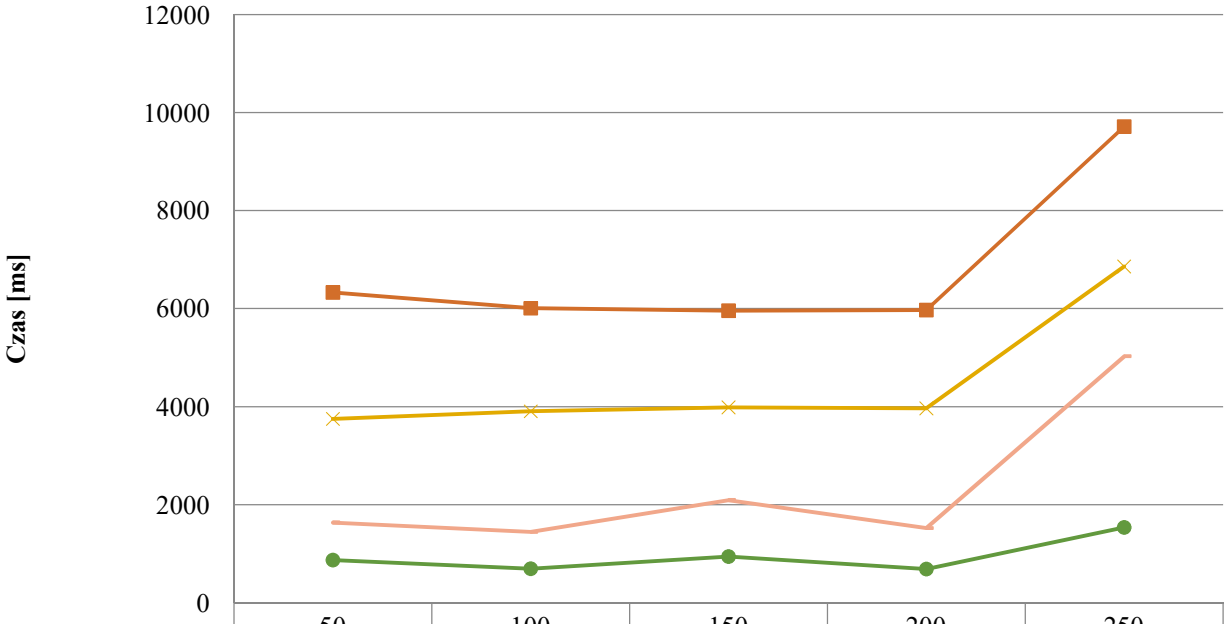
## Gęstość 50% - Lista




| | 50 | 100 | 150 | 200 | 250 |
|---|---|---|---|---|---|
| Prim | 1823 | 1807 | 1768 | 1852 | 2866 |
| Kruskal | 3766 | 3903 | 3735 | 3767 | 7460 |
| Dijkstry | 515 | 446 | 564 | 436 | 862 |
| Bellman-Ford | 201 | 194 | 223 | 180 | 278 |

**Liczba wierzchołków**

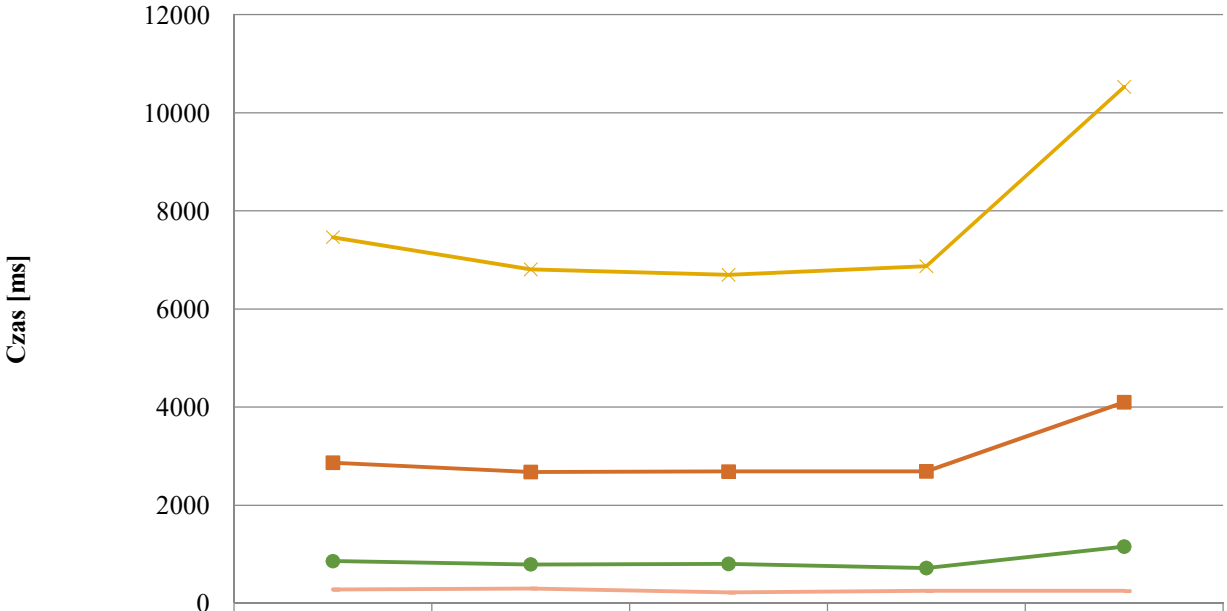
## Gęstość 50% - Macierz




| | 50 | 100 | 150 | 200 | 250 |
|---|---|---|---|---|---|
| Prim | 6332 | 6010 | 5960 | 5976 | 9717 |
| Kruskal | 3753 | 3907 | 3991 | 3971 | 6862 |
| Dijkstry | 870 | 695 | 942 | 687 | 1536 |
| Bellman-Ford | 1635 | 1444 | 2092 | 1523 | 5027 |

**Liczba wierzchołków**

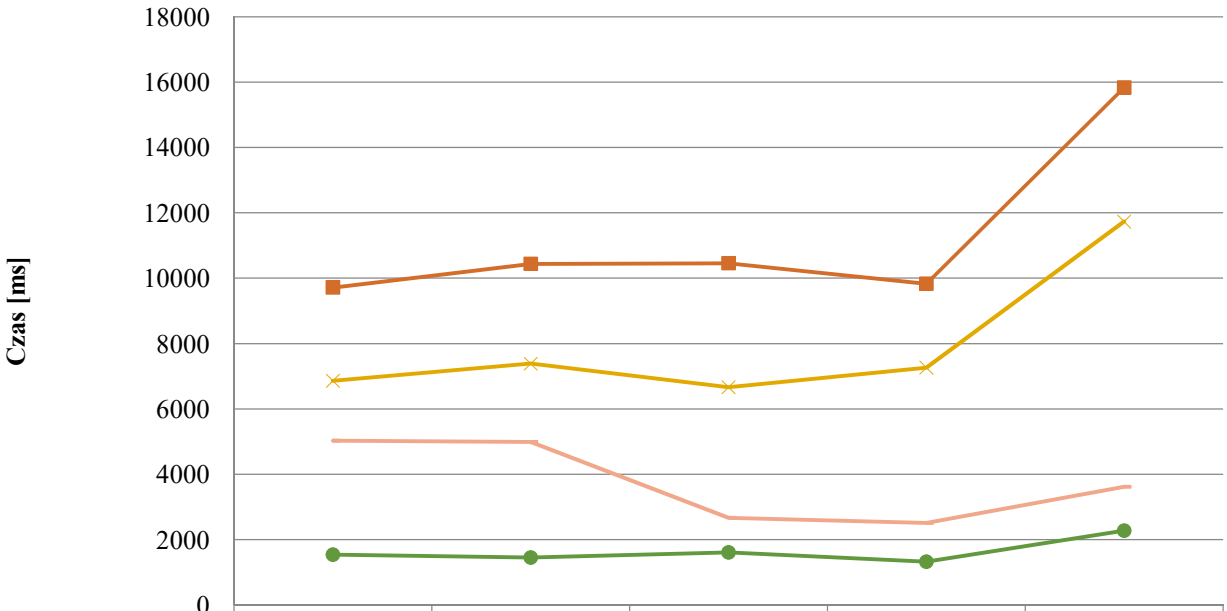
## Gęstość 75% - Lista




| | 50 | 100 | 150 | 200 | 250 |
|---|---|---|---|---|---|
| Prim | 2866 | 2676 | 2687 | 2692 | 4102 |
| Kruskal | 7460 | 6806 | 6693 | 6871 | 10526 |
| Dijkstry | 862 | 796 | 804 | 721 | 1158 |
| Bellman-Ford | 278 | 300 | 220 | 251 | 251 |

**Liczba wierzchołków**

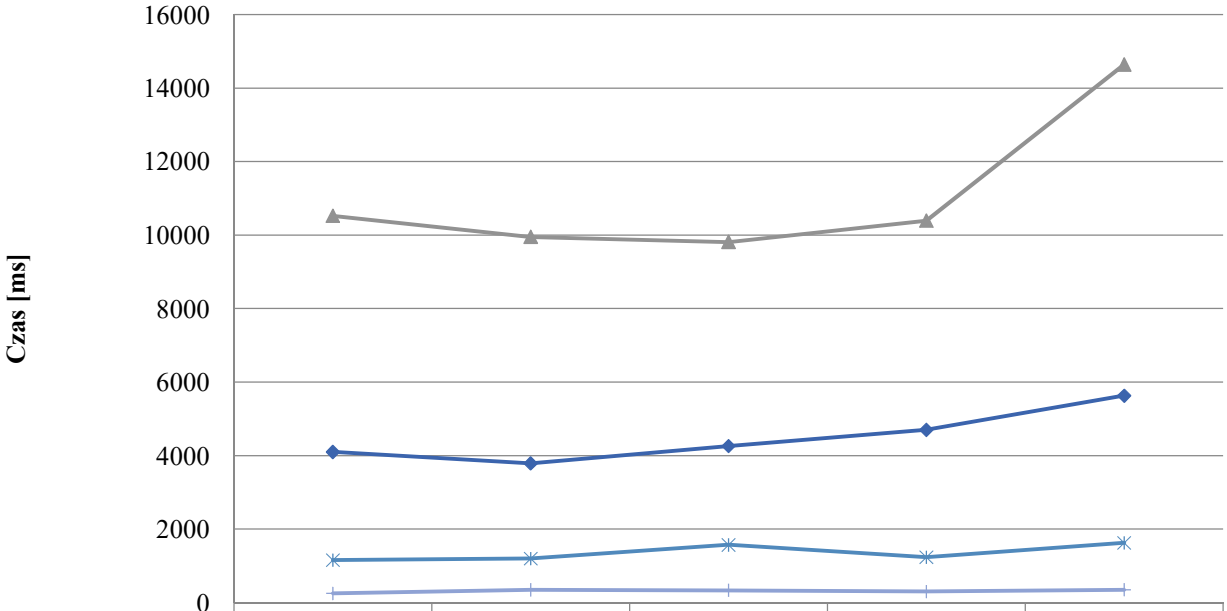
## Gęstość 75% - Macierz




| | 50 | 100 | 150 | 200 | 250 |
|---|---|---|---|---|---|
| Prim | 9717 | 10438 | 10459 | 9832 | 15840 |
| Kruskal | 6862 | 7388 | 6664 | 7264 | 11736 |
| Dijkstry | 1536 | 1451 | 1607 | 1322 | 2273 |
| Bellman-Ford | 5027 | 4987 | 2666 | 2513 | 3618 |

**Liczba wierzchołków**

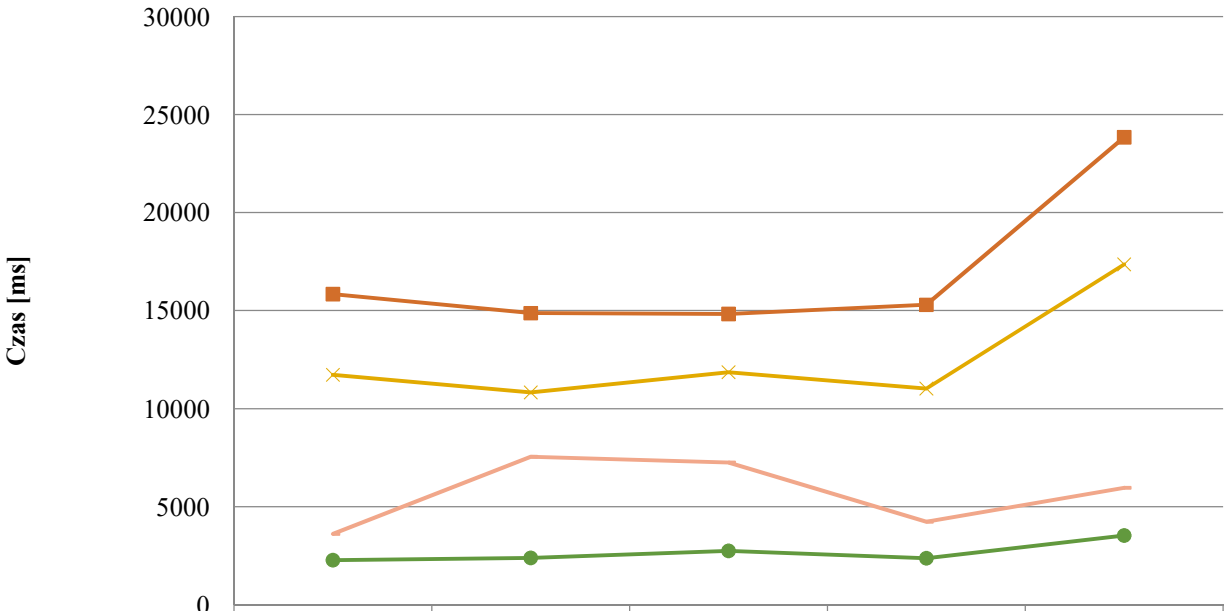
## Gęstość 99% - Lista




| | 50 | 100 | 150 | 200 | 250 |
|---|---|---|---|---|---|
| Prim | 4102 | 3791 | 4263 | 4704 | 5633 |
| Kruskal | 10526 | 9952 | 9813 | 10393 | 14646 |
| Dijkstry | 1158 | 1198 | 1572 | 1237 | 1627 |
| Bellman-Ford | 251 | 348 | 335 | 303 | 352 |

**Liczba wierzchołków**

## Gęstość 99% - Macierz




| | 50 | 100 | 150 | 200 | 250 |
|---|---|---|---|---|---|
| Prim | 15840 | 14867 | 14826 | 15292 | 23843 |
| Kruskal | 11736 | 10832 | 11863 | 11033 | 17361 |
| Dijkstry | 2273 | 2387 | 2749 | 2374 | 3528 |
| Bellman-Ford | 3618 | 7541 | 7251 | 4239 | 5962 |

**Liczba wierzchołków**

## 5. Wnioski

Eksperyment pozwolił nam zaobserwować, jak przedstawiają się wyniki rzeczywistych pomiarów w porównaniu z wcześniejszymi założeniami co do ich złożoności. Powyższe wykresy ukazują, że teorie na temat złożoności są słuszne. Pewne nieścisłości, które pojawiły się w obliczeniach, wynikają prawdopodobne z używania komputera, którego procesor oprócz naszego eksperymentu ma jeszcze inne zadania do wykonania w tym samym czasie. Procesy działające w tle mogły wpływać na chwilowe spadki mocy obliczeniowej i tym samym spowodować wolniejsze wykonywanie operacji na strukturach. Innym powodem mógł być fakt, że program generował losowy zestaw danych, zatem liczby, na których wykonywał operacje, mogły być większe lub mniejsze, co również miało wpływ na pomiar czasu. W większości zaprezentowanych pomiarów zauważamy, że czas wykonania algorytmów opartych o reprezentację listową jest niższy. Dzieje się tak ze względu na krótszy czas alokacji listy w stosunku do macierzy incydencji. Rozbieżności pomiędzy danymi literaturowymi, a przedstawionymi wynikami eksperymentu mogą wynikać z niedokładności pomiaru czasu w systemie Windows, różnej zajętości procesora podczas wykonywania algorytmów, niedoskonałej implementacji struktur (lista, tablica, kopiec, kolejka, stos) oraz, przede wszystkim, wpływających na efektywność błędów w implementacji algorytmów. Rzeczywiste kształty wykresów mogą nie być widoczne przy tak małej ilości danych. Algorytm Kruskala działa podobnie zarówno przy implementacji macierzowej jak i listowej. Oczywiście przy większych gęstościach, potrzeba więcej czasu, aby algorytm się wykonał. W przypadku, kiedy liczy się czas wykonywania operacji, implementacja macierzowa jest lepszym wyborem, jednakże dla niewielkich liczb krawędzi przy stosunkowo dużej liczbie wierzchołków implementacja listowa może być preferowana ze względu na mniejszą złożoność pamięciową. Reprezentacja listowa jest jedną z lepszych reprezentacji - minimalna możliwa złożoność pamięciowa, szybkie przeszukiwanie krawędzi wychodzących z danego wierzchołka, stosunkowo łatwa implementacja. Wadą jest jedynie długi czas sprawdzenia istnienia pojedynczej krawędzi. Można go poprawić do $O(log\ V)$ wprowadzając drzewa BST (albo nawet drzewa zrównoważone) zamiast list, ale to skomplikuje implementację. Należy również pamiętać, że w wykonywanych zadaniach nie została uwzględniona złożoność pamięciowa, która z pewnością pomogłaby w lepszym zobrazowaniu zalet i wad danych rozwiązań.