POLITECHNIKA WARSZAWSKA

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH

ANALIZA ALGORYTMÓW MST II

Autorzy:
Joanna Maciak
Kacper Dominik

Prowadzący: dr inż. Sebastian Kozłowski

Warszawa 2018

Spis treści

1. Założenia projektowe

Tematem projektu jest **przeprowadzenie eksperymentalnej analizy czasu wykonywania dwóch algorytmów wyznaczania najlżejszego drzewa rozpinającego dla różnych klas grafów**.

Wybranymi algorytmami, będącymi przedmiotem tematu niniejszego projektu, są:
– Algorytm Kruskala
– Algorytm Prima

Językiem programowania, wybranym do implementacji powyższych algorytmów jest język Java.

2. Opis elementów projektu

Problem (zdefiniowany na podstawie [2]):
Znalezienie takiego podzbioru krawędzi spójnego grafu nieskierowanego ważonego, który zapewnia połączenie każdego wierzchołka grafu z dowolnym innym i ponadto posiada najmniejszą możliwą sumę wag krawędzi.
Taki podzbiór nie może zawierać żadnego cyklu, a zatem jest drzewem i musi zawierać dokładnie *n-1* krawędzi dla grafu o *n* wierzchołkach. Drzewo takie ze względu na minimalną sumę wag nazywane jest minimalnym drzewem rozpinającym.

2.1. Algorytm Prima

Dane wejściowe:
Spójny ważony graf nieskierowany, zawierający n wierzchołków, gdzie $n \geqslant 2$.
prim2

2.1.1. Pseudokod zaimplementowanego algorytmu

Pseudokod oraz implementację opracowano na podstawie [2] oraz [1].

1. Inicjalizacja
   - Utwórz pusty stos krawędzi MST: *minSpanningTree*.
   - Zainicjalizuj pustą kolejkę priorytetową krawędzi osiągalnych przez MST: *adjacencyEdges*
   - Utwórz tablicę odwiedzonych wierzchołków grafu i ustaw wartości jej pól na *false*
2. Losuj wierzchołek początkowy: *startVertex*.
3. Oznacz wierzchołek *startVertex* jako odwiedzony poprzez zmianę pola tablicy o indeksie *startVertex* na *true*

4. Dodaj do kolejki priorytetowej *adjacencyEdges* krawędzie incydentne do odwiedzonego wierzchołka początkowego

5. Powtarzaj dopóki kolejka priorytetowa *adjacencyEdges* nie będzie pusta:

    a. Pobierz z kolejki *adjacencyEdges* krawędź o najniższym koszcie

    b. Jeśli wierzchołki początkowy *startVertex* i końcowy *endVertex* pobranej krawędzi zostały już odwiedzone, to pomiń tę iterację i przejdź do podpunktu **a.** W przeciwnym wypadku przejdź do **c.**

    c. Dodaj pobraną krawędź do drzewa *minSpanningTree*

    d. Jeśli wierzchołek początkowy pobranej krawędzi nie został odwiedzony, to oznacz go jako odwiedzony i dodaj do kolejki *adjacencyEdges* krawędzie incydentne do tego wierzchołka

    e. Jeśli wierzchołek końcowy pobranej krawędzi nie został odwiedzony, to oznacz go jako odwiedzony i dodaj do kolejki *adjacencyEdges* krawędzie incydentne do tego wierzchołka

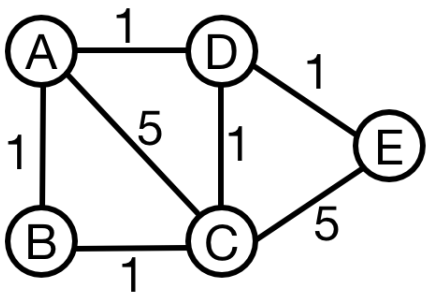
6. Jeśli kolejka priorytetowa już nie zawiera krawędzi, to otrzymany stos *minSpanningTree* reprezentuje otrzymane minimalne drzewo rozpinające MST

2.1.2. Zasada działania zaimplementowanego algorytmu na przykładzie

Poniżej znajduje się legenda oznaczeń wykorzystanych w opisie działania algorytmu na przykładzie.

rys. 2.1. Oznaczenia wykorzystane w opisie

Dany jest graf widoczny na rys.2.2. Wierzchołki oznaczono wielkimi literami alfabetu, natomiast krawędziom nadano wagi $w \subseteq \{1,5\}$.



rys. 2.2. Graf wejściowy.

Inicjalizacja

Stos: *minSpanningTree* ={ };

Tablica wierzchołków odwiedzonych: *visitedVertices* = [false, false, false, false, false];

Kolejka priorytetowa krawędzi incydentnych: *adjacencyEdges* = [ ];

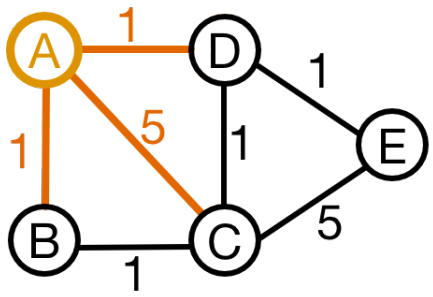
Krok 1.

Losowanie wierzchołka początkowego. W tym przypadku niech będzie to wierzchołek **A**.

startVertex = A

Krok 2.

Oznaczenie wierzchołka *startVertex* jako odwiedzony oraz dodanie jego krawędzi incydentnych do kolejki priorytetowej.



rys. 2.3. Prim – Krok 2.

Stos: *minSpanningTree* ={ };

Tablica wierzchołków odwiedzonych: *visitedVertices* = [true, false, false, false, false];

Kolejka priorytetowa krawędzi incydentnych: *adjacencyEdges* = [(A - D), (A - B), (A - C)];

Krok 3.

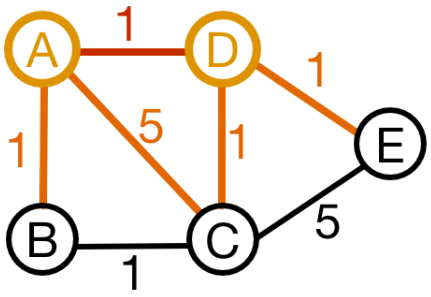
Pobranie z kolejki krawędzi (A - D). Jej wierzchołek końcowy nie jest jeszcze odwiedzony, więc krawędź ta zostaje dodana do *minSpanningTree*. Wierzchołek końcowy tej krawędzi - D - zostaje oznaczony jako odwiedzony, a jego krawędzie incydentne zostają dodane do kolejki *adjacencyEdges* .

Stos: *minSpanningTree* ={ (A - D)};

Tablica wierzchołków odwiedzonych: *visitedVertices* = [true, false, false, true, false];

Kolejka priorytetowa krawędzi incydentnych:

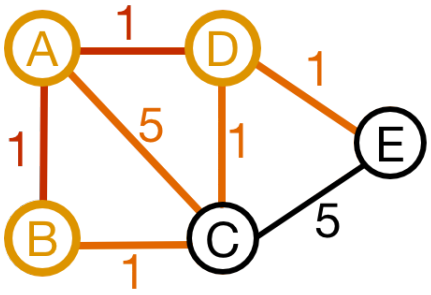
adjacencyEdges = [(A - B), (D - E), (D - C), (A - C)];



rys. 2.4. Prim – Krok 3.

Krok 4.

Pobranie z kolejki krawędzi (A - B). Jej wierzchołek końcowy nie jest jeszcze odwiedzony, więc krawędź ta zostaje dodana do *minSpanningTree*. Wierzchołek końcowy tej krawędzi - B - zostaje oznaczony jako odwiedzony, a jego krawędzie incydentne zostają dodane do kolejki *adjacencyEdges* .

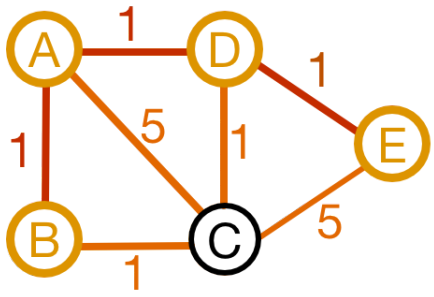


rys. 2.5. Prim – Krok 4.

Stos: *minSpanningTree* ={ (A - B), (A - D) };
Tablica wierzchołków odwiedzonych: *visitedVertices* = [true , true, false, true, false];
Kolejka priorytetowa krawędzi incydentnych:
adjacencyEdges = [(D - E), (D - C), (B - C), (A - C)];

Krok 5.

Pobranie z kolejki krawędzi (D - E). Jej wierzchołek końcowy nie jest jeszcze odwiedzony, więc krawędź ta zostaje dodana do *minSpanningTree*. Wierzchołek końcowy tej krawędzi - E - zostaje oznaczony jako odwiedzony, a jego krawędzie incydentne zostają dodane do kolejki *adjacencyEdges* .

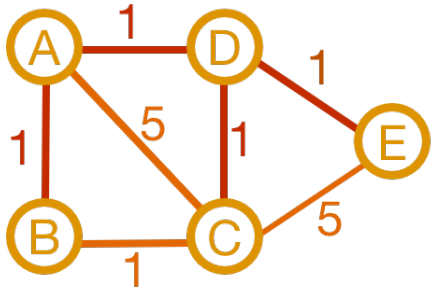


rys. 2.6. Prim – Krok 5.

Stos: *minSpanningTree* ={ (D - E), (A - B), (A - D) };
Tablica wierzchołków odwiedzonych: *visitedVertices* = [true , true, false, true, true];
Kolejka priorytetowa krawędzi incydentnych:
adjacencyEdges = [(D - C), (B - C), (A – C), (E - C)];

Krok 6.

Pobranie z kolejki krawędzi (D - C). Jej wierzchołek końcowy nie jest jeszcze odwiedzony, więc krawędź ta zostaje dodana do *minSpanningTree*. Wierzchołek końcowy tej krawędzi - C - zostaje oznaczony jako odwiedzony. Jego krawędzie incydentne nie zostają jednak dodane do kolejki *adjacencyEdges*, ponieważ ich wierzchołki końcowe zostały już odwiedzone.



rys. 2.7. Prim – Krok 6.

Stos: *minSpanningTree* ={ (D - C), (D - E), (A - B), (A - D) };
Tablica wierzchołków odwiedzonych: *visitedVertices* = [true , true, true, true, true];
Kolejka priorytetowa krawędzi incydentnych: *adjacencyEdges* = [(B - C), (A – C), (E - C)];

Krok 7., 8., 9.

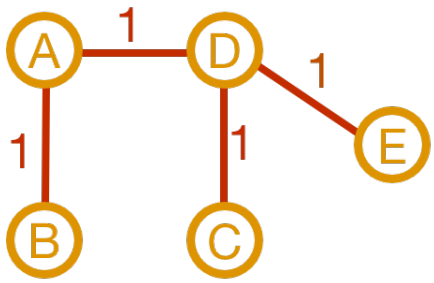
Pobranie z kolejki *adjacencyEdges* krawędzi w kolejności : (B - C), (A – C), (E - C). Ich wierzchołki zarówno początkowe jak i końcowe zostały już odwiedzone, w związku z czym pomija się dalszą część iteracji.

Stos: *minSpanningTree* ={ (A - D), (A - B), (D - E), (D - C) };
Tablica wierzchołków odwiedzonych: *visitedVertices* = [true , true, true, true, true];
Kolejka priorytetowa krawędzi incydentnych: *adjacencyEdges* = [(B - C), (A – C), (E - C)];

Krok 10.

Kolejka priorytetowa krawędzi incydentnych jest pusta. Otrzymane MST wygląda następująco:

minSpanningTree ={ (A - D), (A - B), (D - E), (D - C) }



rys. 2.8. Prim – Krok 7.

2.2. Algorytm Kruskala

2.2.1. Pseudokod opisujący działanie algorytmu Kruskala:

Inicjalizacja:

1. Utwórz las L z wierzchołków oryginalnego grafu – każdy wierzchołek jest na początku osobnym drzewem.
2. Utwórz posortowany zbiór S zawierający wszystkie krawędzie oryginalnego grafu.

Tworzenie MST

1. Wybierz i usuń z S jedną z krawędzi o minimalnej wadze.
2. Jeśli krawędź ta łączyła dwa różne drzewa, to dodaj ją do lasu L tak, aby połączyła dwa odpowiadające drzewa w jedno.
3. W przeciwnym wypadku odrzuć ją.
4. Jeżeli wciąż istnieje więcej niż jedno drzewo przejdź do punktu 1.

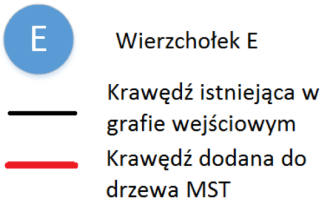
Po zakończeniu algorytmu L jest minimalnym drzewem rozpinającym zgodnie z [3].

2.2.2. Działanie algorytmu na przykładzie

Dany jest graf posiadający następujące składowe:

- Wierzchołki: A, B, C, D, E
- Krawędzie: A B, 1, A D, 1, C D, 1, D E, 1, B C, 1, A C, 5, C E, 5

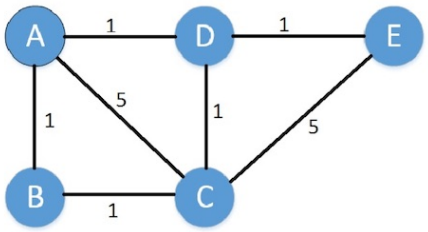
Legenda oznaczeń w poniższym przykładzie:



rys. 2.9. Oznaczenia wykorzystane w opisie algorytmu Kruskala

Przebieg algorytmu

1. Pierwszy etap wykonania algorytmu - ustalenie zbioru wierzchołków S oraz zbioru krawędzi L:
   S: A, B, C, D, E
   L: A B, A D, C D, D E, B C, A C, C E



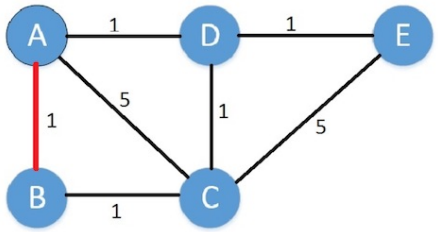
rys. 2.10. Przykładowy graf / pierwszy etap wykonania algorytmu Kruskala

2. Krok 2

   Ze zbioru S usunięta została jedna z krawędzi o najmniejszej wadze. A B Ponieważ wierzchołki, które łączyła znajdowały się w osobnych drzewach w L zostały połączone w jedno drzewo A, B, więc dodawana jest krawędź dodawana jest do W.

   S: A, B, C, D, E

   L: A D, C D, D E, B C, A C, C E



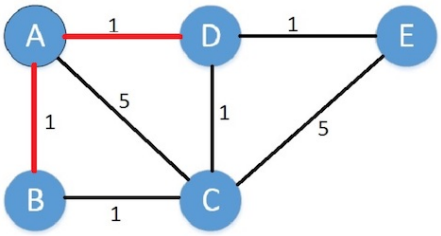
rys. 2.11. Drugi etap wykonania algorytmu Kruskala

3. Krok 3

   Ze zbioru S usunięta została jedna z krawędzi o najmniejszej wadze. A D Ponieważ wierzchołki, które łączyła znajdowały się w osobnych drzewach w L zostały połączone w jedno drzewo A, B, D, więc dodawana jest krawędź dodawana jest do W.

   S: A, B, D, C, E

   L: C D, D E, B C, A C, C E



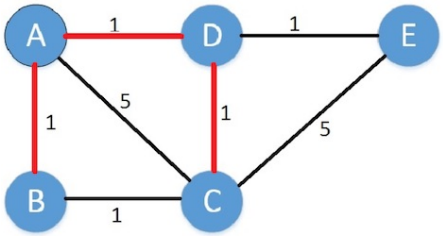
rys. 2.12. Trzeci etap wykonania algorytmu Kruskala

4. Krok 4

   Ze zbioru S usunięta została jedna z krawędzi o najmniejszej wadze. C D Ponieważ wierzchołki, które łączyła znajdowały się w osobnych drzewach w L zostały połączone w jedno drzewo A, B, C, D, więc dodawana jest krawędź dodawana jest do W.

   S: A, B, C, D, E

   L: D E, B C, A C, C E



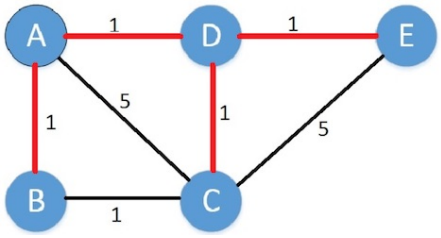
rys. 2.13. Czwarty etap wykonania algorytmu Kruskala

5. Krok 5

   Ze zbioru S usunięta została jedna z krawędzi o najmniejszej wadze. D E Ponieważ wierzchołki, które łączyła znajdowały się w osobnych drzewach w L zostały połączone w jedno drzewo A, B, C, D, E, więc dodawana jest krawędź dodawana jest do W. Ponieważ w zbiorze L pozostał tylko jeden zbiór wykonanie algorytmu zostaje zakończone.

   S: A, B, C, D, E

   L: B C, A C, C E



rys. 2.14. Piąty etap wykonania algorytmu Kruskala – uzyskane MST

Jak widać nie zawiera ono żadnych cykli i zawiera tylko krawędzie o minimalnej wadze.

2.3. Opis wybranych struktur danych

Aby uzyskać taką samą złożoność czasową dla obu wybranych algorytmów ($O(E \cdot log\ V)$), obrano następujące założenia:

- Implementacja algorytmu Prima jest oparta na kolejce priorytetowej, w której operacja wstawiania elementu ma złożoność $O(logn)$, gdzie n, to liczba elementów w kolejce.
- Implementacja algorytmu Kruskala jest oparta na listowej implementacji struktury zbiorów rozłącznych.

Aby zrealizować ten cel, wykorzystano w implementacji obu algorytmów następujące struktury danych:

2.3.1. Stos

Stos jest strukturą danych, z której elementy są odczytywane w kolejności odwrotnej do ich wstawiania. Struktura ta nosi nazwę LIFO. Rozróżniamy następujące operacje dla stosu:

- sprawdzenie, czy stos jest pusty – operacja **empty** zwraca true, jeśli stos nie zawiera żadnego elementu, w przeciwnym razie zwraca false
- odczyt szczytu stosu – operacja **top** zwraca element (zwykle jest to wskaźnik) znajdujący się na szczycie stosu, sam element pozostaje wciąż na stosie
- zapis na stos – operacja **push** umieszcza na szczycie stosu nowy element
- usunięcie ze stosu – operacja **pop** usuwa ze szczytu stosu znajdujący się tam element

Stos krawędzi w implementacji algorytmu Prima reprezentuje minimalne drzewo rozpinające - MST i krawędzie są jedynie do niego dodawane metodą **push**.

2.3.2. Kolejka priorytetowa

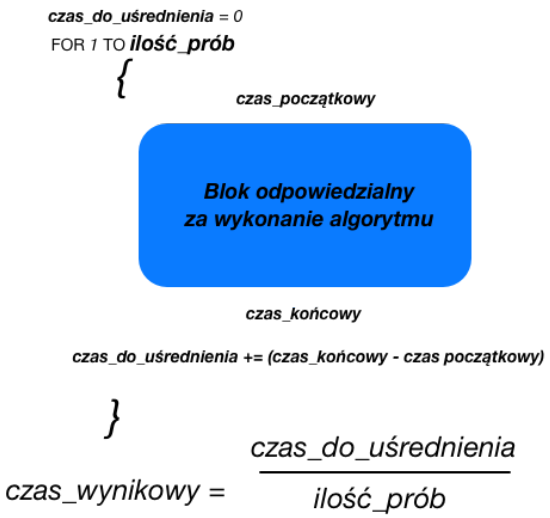
Kolejka priorytetowa jest kolejką, w której elementy są ułożone nie w kolejności wprowadzania, lecz w kolejności priorytetu. Każdy element kolejki posiada dodatkowe pole *prio*, w którym przechowuje swój priorytet – czyli ważność. Gwarantuje to pobieranie z kolejki jako pierwszych elementów o najwyższym priorytecie. Elementy o priorytetach niższych zostaną pobrane dopiero wtedy, gdy zostaną usunięte wszystkie elementy o priorytetach wyższych.
W implementacji algorytmu Prima rolę *prio* pełni waga krawędzi.
W niniejszym projekcie operacje **empty**, **front** i **pop** niczym się nie różnią od takich samych operacji dla zwykłej kolejki. Jedyna różnica pojawia się przy operacji *push*, która umieszcza w kolejce nowy element. W tym przypadku lista jest przeglądana od początku do końca, aby znaleźć w niej krawędź o wadze bezpośrednio wyższej od wagi dodawanej krawędzi. Przed znalezioną krawędzią zostaje dodana nowa krawędź.

2.3.3. Tablica

Tablica, to kontener danych takiego samego typu, gdzie każdy z elementów jest dostępny za pomocą klucza zwanego indeksem. W implementacji algorytmu Prima, tablica *visitedVertices* reprezentuje odwiedzone wierzchołki grafu i jest to tablica jednowymiarowa.

2.4. Założenia programu i ogólny projekt testów

1. Algorytmy zostaną wykonane dla **grafów spójnych, nieskierowanych**.
2. Wszystkie pomiary zostaną wykonane na następującej maszynie:
   – MacBook Air z procesorem 1,7 GHz Intel Core i7
3. Porównaniu ulegnie czas wykonywania obu algorytmów.
4. Oba algorytmy zostaną wykonane dla grafów o określonej liczbie wierzchołków oraz krawędzi.
5. Pomiar czasu będzie wykonywany od momentu oznaczającego pierwszy krok każdego z algorytmów aż do wykonania ostatniej istrukcji z ostatniego kroku.
6. Pomiar czasu będzie wykonywany w około 1000 iteracjach, a następnie ulegnie uśrednieniu. Schemat poglądowy programu podczas pomiaru czasu wygląda następująco:



rys. 2.15. Zarys pomiaru czasu podczas wykonywania algorytmu

Dokładna liczba iteracji zostanie dobrana metodą prób i błędów podczas wykonywania pomiarów.

3. Przebieg badań

3.1. Obrane założenia

1. Przez wzgląd na duży rozmiar wygenerowanych grafów wybranych do wykonania projektu, w celu przeprowadzenia obliczeń wykorzystano **MacBook Pro z procesorem Intel Core i7-6820HQ oraz 16 GB pamięci RAM**.

2. Minimalna liczba iteracji, wystarczająca do wykonania badań, to **100**, jednak przez wzgląd na możliwości sprzętowe, zdecydowano się na liczbę iteracji równą **1000**.

3. Badania przeprowadzono dla grafów o **liczbie wierzchołków** równej: 10, 50, 100, 500, 1000, 5000 oraz 10 000.

4. Badania przeprowadzono dla grafów o **gęstości** równej: 0.2, 0.4, 0.6, 0.8 oraz 1.0.

3.2. Przebieg obliczeń

Aby określić liczbę iteracji, przeprowadzono testy na grafie pełnym o 1000 wierzchołkach. Wyniki tych badań prezentuje tabela 3.1.

PRIM	
Ilość iteracji	**Czas [µs]**
10	453100
50	348640
100	390330
200	393400
300	369893
500	384740
1000	382426
KRUSKAL	
Ilość iteracji	**Czas [µs]**
10	95400
50	59780
100	60490
200	60275
300	60893
500	58594
1000	58321

rys. 3.1. Wyniki badań czasu wykonania algorytmów dla różnej liczby iteracji.

Z tabeli 3.1 wynika, że minimalną liczbą iteracji dla prowadzonych obliczeń jest liczba 100, ponieważ wartości czasu wykonania algorytmu dla większych od niej liczb iteracji niewiele się od siebie różnią. **Jednak dzięki dostępnym możliwościom sprzętowym zdecydowano się na ustawienie liczby iteracji równej 1000.**

Następnie obliczono liczbę krawędzi dla grafów o danej liczbie wierzchołków oraz zakładanej gęstości. Wyniki obliczeń, zarazem zbiór danych grafów wejściowych, prezentuje tabela 3.2.

Liczba wierzchołków	Liczba krawędzi	Gęstość grafu
10	18	0,4
10	27	0,6
10	36	0,8
10	45	1
50	245	0,2
50	490	0,4
50	735	0,6
50	980	0,8
50	1225	1
100	990	0,2
100	1980	0,4
100	2970	0,6
100	3960	0,8
100	4950	1
500	24950	0,2
500	49900	0,4
500	74850	0,6
500	99800	0,8
500	124750	1
1000	99900	0,2
1000	199800	0,4
1000	299700	0,6
1000	399600	0,8
1000	499500	1
5000	2499500	0,2
5000	4999000	0,4
5000	7498500	0,6
5000	9998000	0,8
5000	12497500	1
10000	9999000	0,2
10000	19998000	0,4
10000	29997000	0,6
10000	39996000	0,8
10000	49995000	1

rys. 3.2. Dane niezbędne do wygenerowania grafu o określonych parametrach.

Po ustawieniu liczby iteracji w programie, przystąpiono do wykonania obliczeń w następujący sposób:

1. Wygenerowano graf o zadanych parametrach.
2. Po wygenerowaniu grafu uruchomiono algorytm Prima.
3. Po wykonaniu wszystkich iteracji dla algorytmu Prima, uruchomiono algorytm Kruskala.

3.2.1. Weryfikacja poprawności implementacji algorytmów

Algorytmy zostały przetestowane na grafach dla których znane jest poprawne MST. Następnie porównane zostały wyniki obu algorytmów dla pewnych grafów losowych. Podczas wszystkich przeprowadzonych testów liczone i porównywane były nie tylko zbiory krawędzi tworzących MST, ale również sumy wag tych krawędzi.

Wstępne obliczenia dla grafu pełnego o 1000 wierzchołkach wykazały dużą rozbieżność pomiędzy wynikami czasu wykonania dla obu algorytmów (na niekorzyść Prima). Wpływ na zaistniałą sytuację miały następujące czynniki:

1. Niewłaściwy sposób pomiaru czasu dla algorytmu Kruskala - pominięto sortowanie krawędzi na początku algorytmu.
2. Struktura reprezentująca graf – algorytm Prima zoptymalizowano dzięki wykorzystaniu do tego celu list sąsiedztwa (zamiast macierzy sąsiedztwa).

Wyniki przeprowadzonych obliczeń zaprezentowano w sekcji 3.3.

3.3. Wyniki eksperymentu

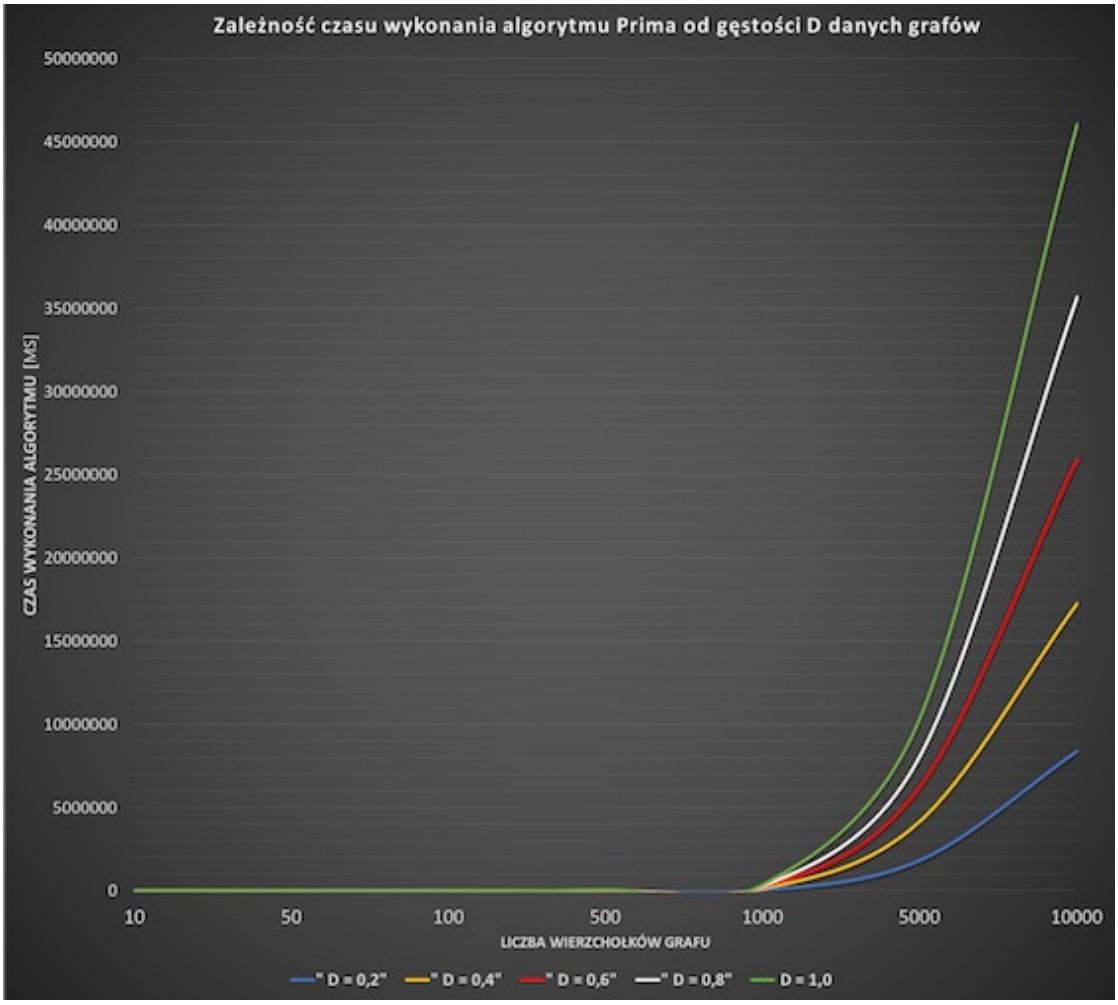
Po dokonaniu optymalizacji i poprawy zaistniałych niedociągnięć wykonano obliczenia, których wyniki zestawiono w tabeli 3.3.

Graf:	O ZADANEJ LICZBIE WIERZCHOŁKÓW I KRAWĘDZI				
Parametry wejściowe				Czas wykonania [mikrosekundy]	
Ilość iteracji	V	E	D	Kruskal	Prim
1000	10	18	0,4	0	30
1000	10	27	0,6	20	20
1000	10	36	0,8	10	20
1000	10	45	1	20	20
1000	50	245	0,2	70	80
1000	50	490	0,4	110	160
1000	50	735	0,6	440	210
1000	50	980	0,8	270	350
1000	50	1225	1	540	440
1000	100	990	0,2	440	280
1000	100	1980	0,4	730	640
1000	100	2970	0,6	540	900
1000	100	3960	0,8	610	970
1000	100	4950	1	630	1220
1000	500	24950	0,2	2470	7230
1000	500	49900	0,4	4560	15570
1000	500	74850	0,6	6570	23940
1000	500	99800	0,8	9070	32820
1000	500	124750	1	11230	53280
1000	1000	99900	0,2	9000	33050
1000	1000	199800	0,4	18910	109250
1000	1000	299700	0,6	34320	195180
1000	1000	399600	0,8	46240	269490
1000	1000	499500	1	56070	361200
1000	5000	2499500	0,2	322050	1836900
1000	5000	4999000	0,4	740490	4170190
1000	5000	7498500	0,6	1443600	6252380
1000	5000	9998000	0,8	1690550	8089660
1000	5000	12497500	1	3779950	10387260
1000	10000	9999000	0,2	2810070	8376260
1000	10000	19998000	0,4	5774240	17245130
1000	10000	29997000	0,6	8377290	25932360
1000	10000	39996000	0,8	10583400	35676530
1000	10000	49995000	1	14050180	46057560

rys. 3.3. Wyniki obliczeń dla obu algorytmów.

Otrzymane wyniki przedstawiono również w formie wykresów:

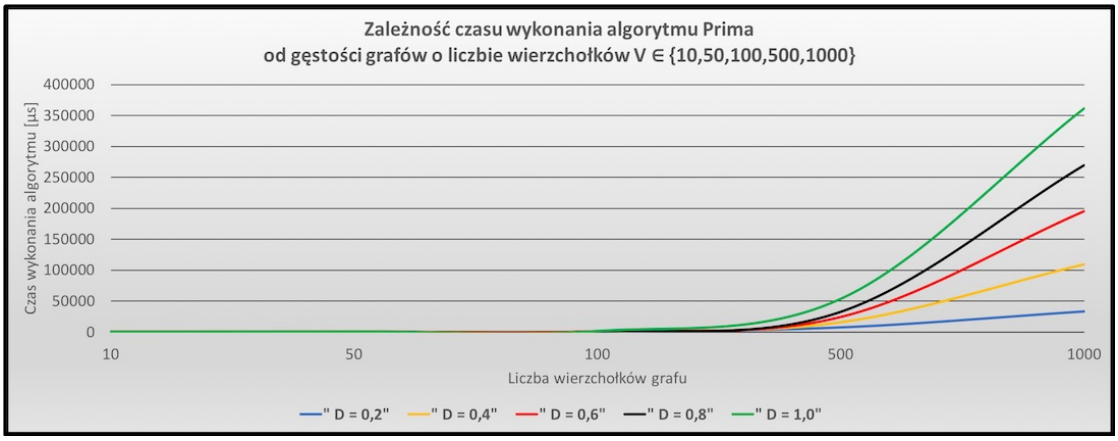
- Zależność czasu wykonania algorytmu Prima od gęstości D danych grafów prezentuje wykres 3.4.



rys. 3.4. Zależność czasu wykonania algorytmu Prima od gęstości D danych grafów.

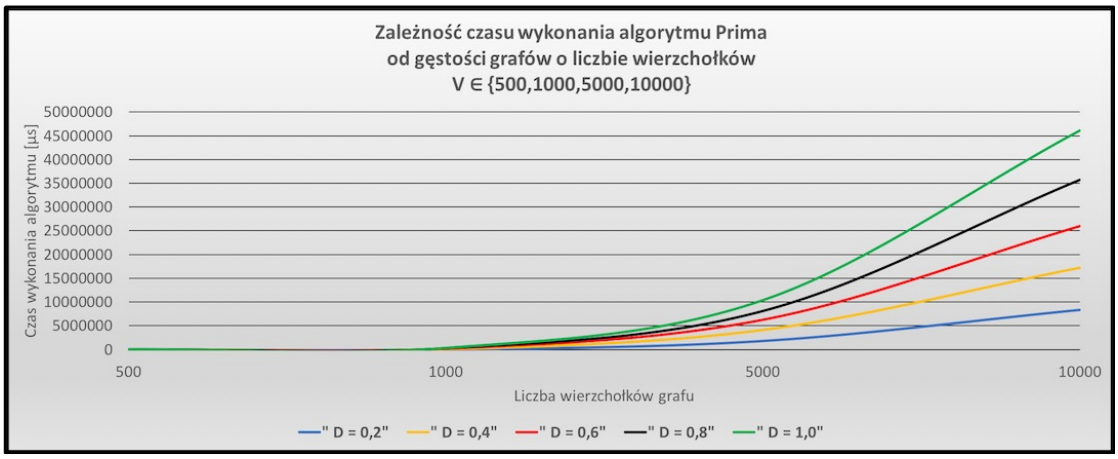
Wykres ten został też podzielony na dwie części w celu zwiększenia czytelności:

1. Zależność czasu wykonania algorytmu Prima od gęstości grafów o liczbie wierzchołków V⊆{10,50,100,500,1000} - rys. 3.5



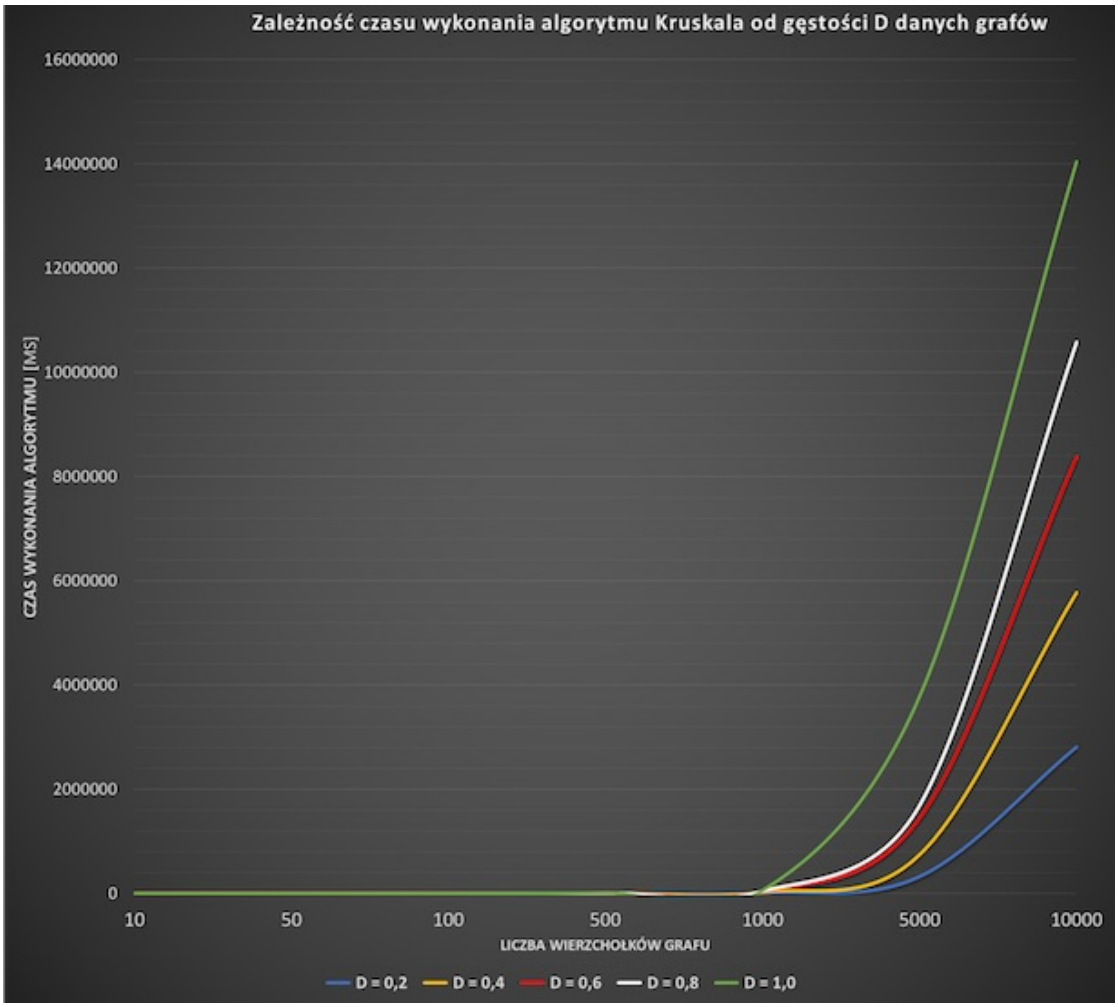
rys. 3.5. Zależność czasu wykonania algorytmu Prima od gęstości grafów o liczbie wierzchołków V⊆ {10,50,100,500,1000}

2. Zależność czasu wykonania algorytmu Prima od gęstości grafów o liczbie wierzchołków V⊆ {500,1000,5000,10000} - rys. 3.6.



rys. 3.6. Zależność czasu wykonania algorytmu Prima od gęstości grafów o liczbie wierzchołków V⊆ {500,1000,5000,10000}

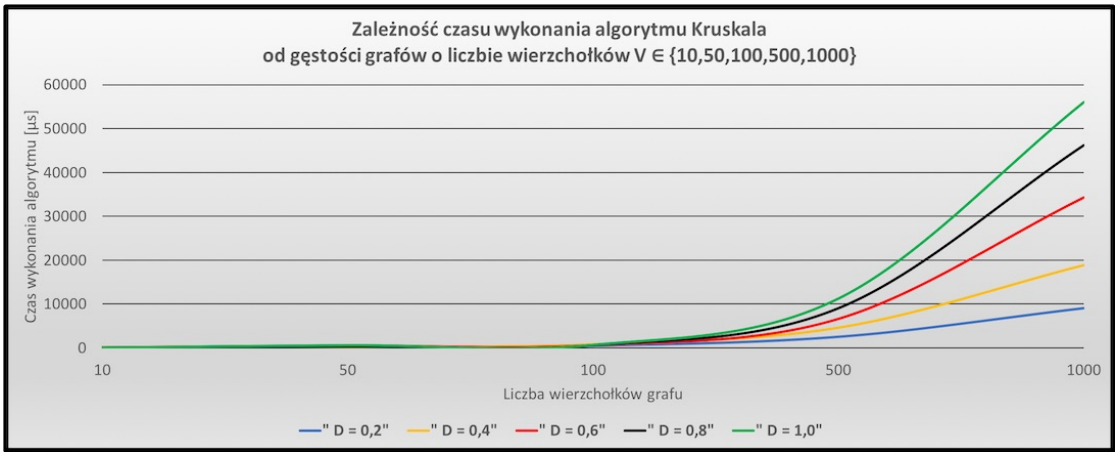
- Zależność czasu wykonania algorytmu Kruskala od gęstości D danych grafów prezentuje wykres 3.7.



rys. 3.7. Zależność czasu wykonania algorytmu Kruskala od gęstości D danych grafów.

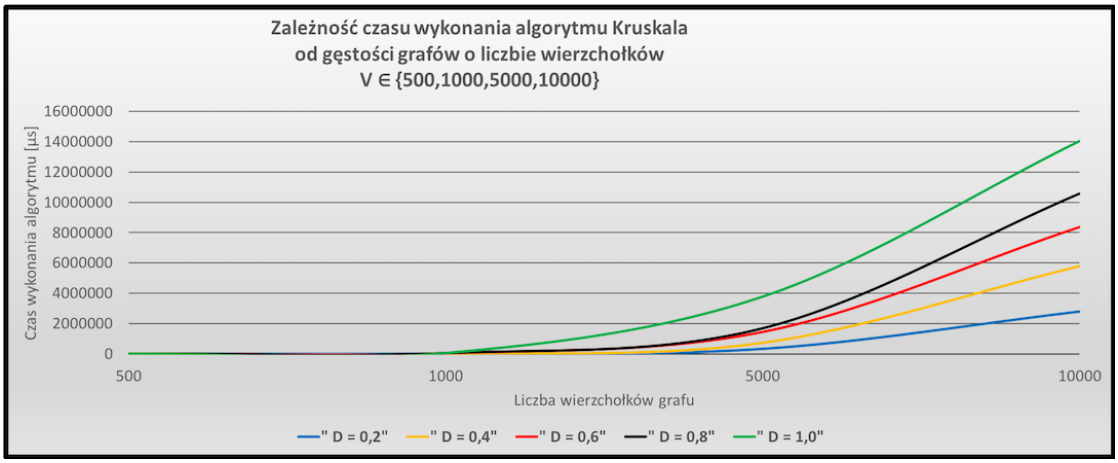
Wykres ten został też podzielony na dwie części w celu zwiększenia czytelności:

1. Zależność czasu wykonania algorytmu Kruskala od gęstości grafów o liczbie wierzchołków V⊆ {10,50,100,500,1000}- rys. 3.8



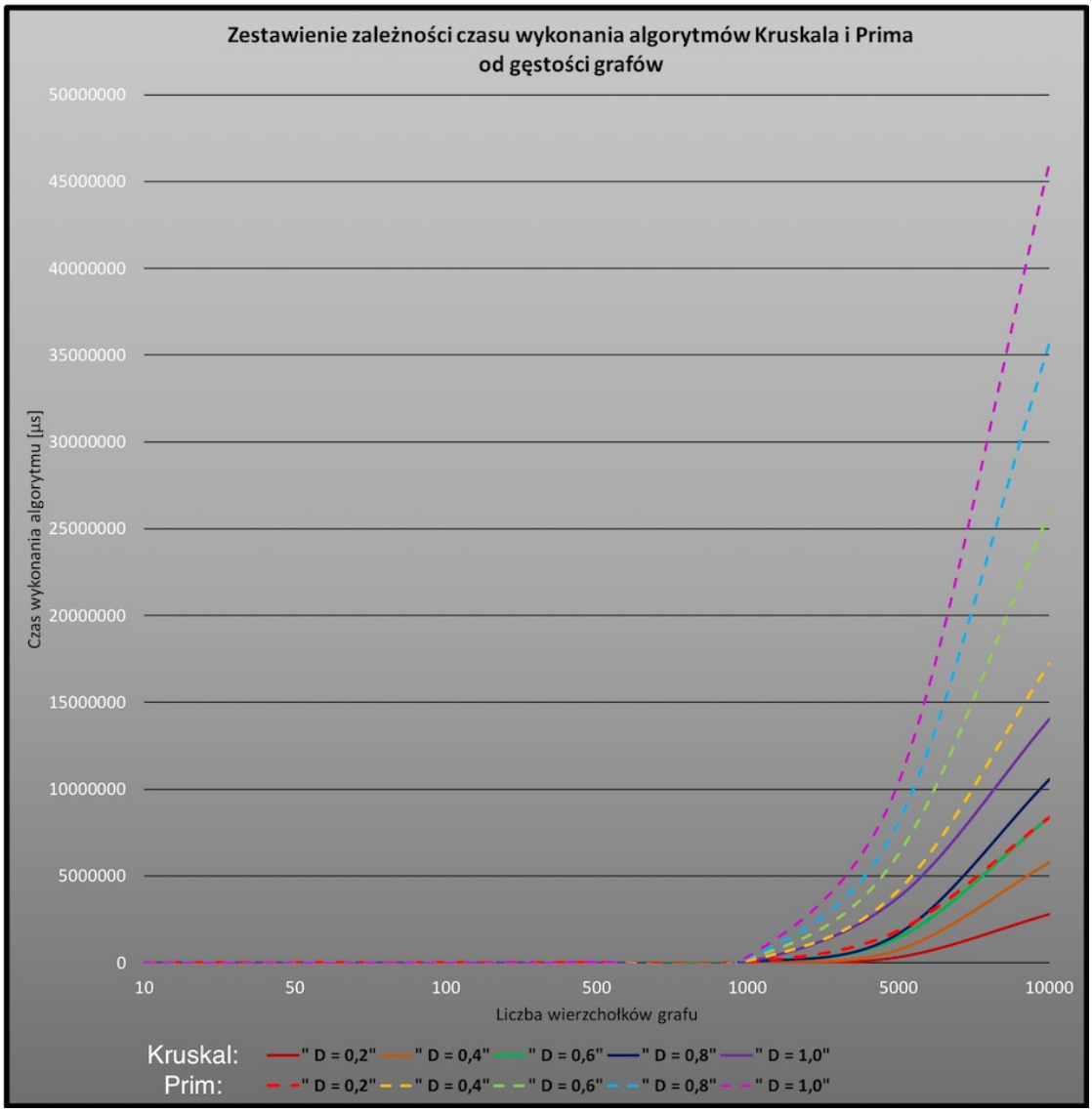
rys. 3.8. Zależność czasu wykonania algorytmu Kruskala od gęstości grafów o liczbie wierzchołków V⊆ {10,50,100,500,1000}

2. Zależność czasu wykonania algorytmu Kruskala od gęstości grafów o liczbie wierzchołków V⊆ {500,1000,5000,10000} - rys. 3.8.



rys. 3.9. Zależność czasu wykonania algorytmu Kruskala od gęstości grafów o liczbie wierzchołków V⊆ {500,1000,5000,10000}

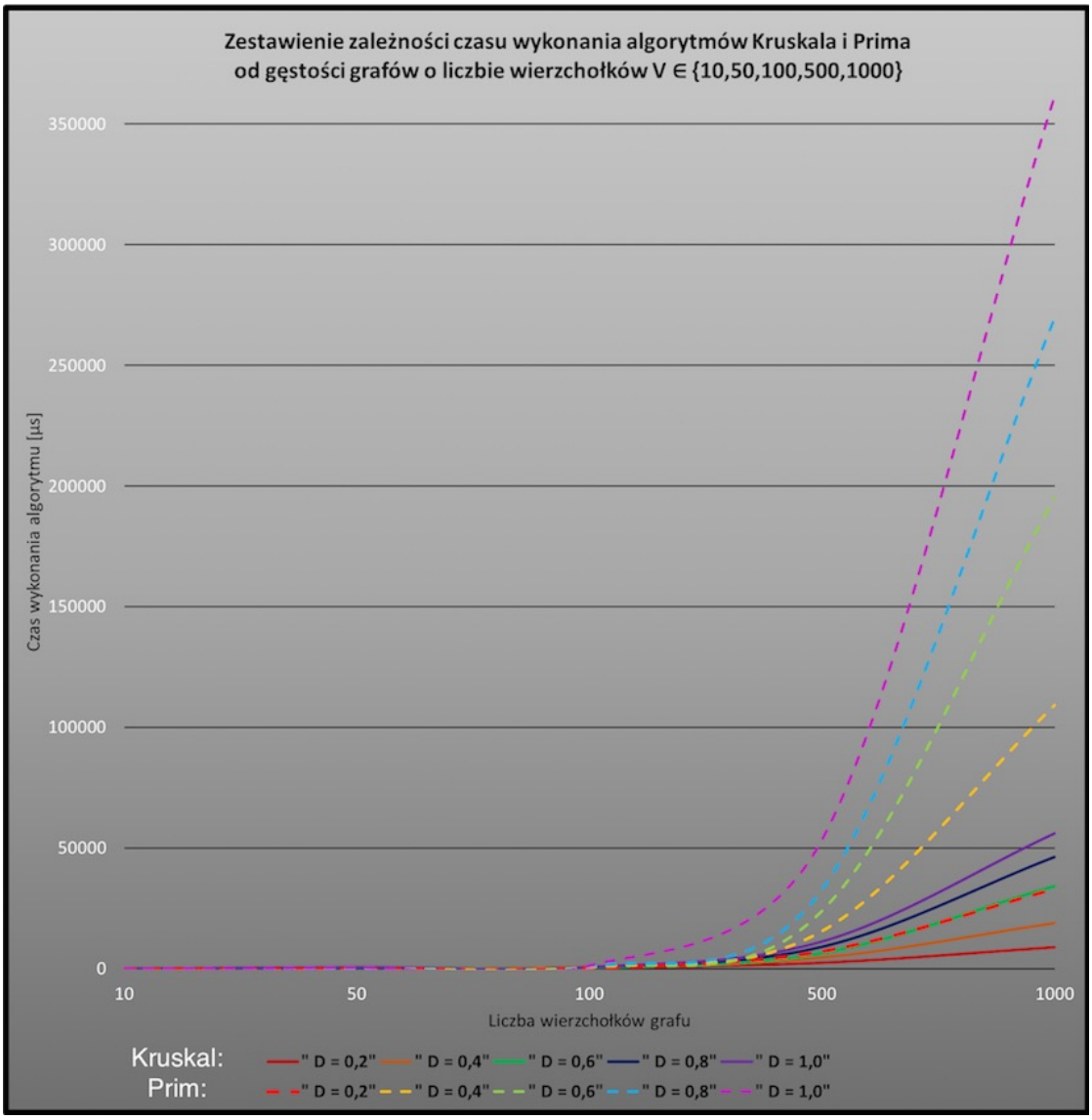
- Zestawienie czasów wykonania algorytmów Prima i Kruskala dla grafów o zadanych parametrach z kolei prezentuje rys. 3.10.



rys. 3.10. Zestawienie zależności czasów wykonania algorytmów Prima i Kruskala od gęstości grafów

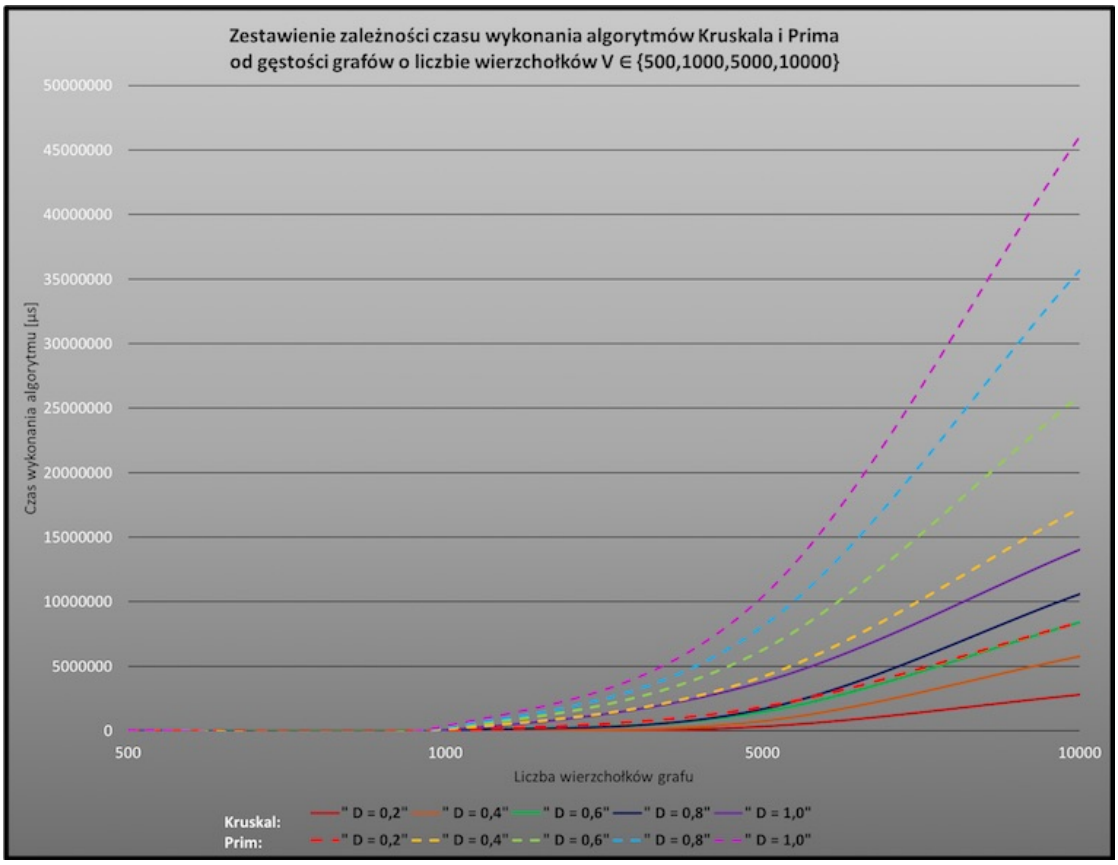
Diagram ten jest podstawą dla wniosków wysuniętych w sekcji 3.4. Dla poprawy czytelności podzielono go na dwie części:

1. Zestawienie zależności czasu wykonania algorytmów Kruskala i Prima od gęstości grafów o liczbie wierzchołków V⊆ {10,50,100,500,1000} - rys. 3.11.



rys. 3.11. Zestawienie zależności czasu wykonania algorytmów Kruskala i Prima od gęstości grafów o liczbie wierzchołków V⊆ {10,50,100,500,1000}

2. Zestawienie zależności czasu wykonania algorytmów Kruskala i Prima od gęstości grafów o liczbie wierzchołków V⊆ {500,1000,5000,10000} - rys. 3.12.



rys. 3.12. Zestawienie zależności czasu wykonania algorytmów Kruskala i Prima od gęstości grafów o liczbie wierzchołków V⊆ {500,1000,5000,10000}

3.4. Wnioski

Na podstawie powyższych wyników obliczeń wyciągnięto następujące wnioski:

1. Dla małych grafów algorytm Prima uzyskuje porównywalne (a nawet lepsze) wyniki względem algorytmu Kruskala
2. W przypadku dużych grafów zastosowanie algorytmu Kruskala sprawdza się lepiej przez wzgląd na krótszy czas wykonywania
3. Z diagramu wynika, że algorytm Prima ma dłuższy czas wykonywania od algorytmu Kruskala
4. W przypadku grafów o małej gęstości algorytm Kruskala radzi sobie znacznie lepiej od algorytmu Prima
5. Różnice czasu wykonywania algorytmu wynikają z:
   – Implementacji każdego z algorytmów
   – Wykorzystanych do implementacji struktur danych
   – Doboru odpowiednich parametrów generowanego do obliczeń grafu

Załączniki

1. Link do repozytorium z kodem źródłowym programu

https://github.com/joannasia9/GenGraf

2. Kod źródłowy algorytmu Prima

Składowe implementacji algorytmu Prima – rys. 3.13

Składowa	Rodzaj	Opis
visitedVertices	boolean[]	Tablica odwiedzonych wierzchołków grafu
minSpanningTree	Deque<PrimEdge>	Stos krawędzi MST
adjacencyEdges	PriorityQueue<PrimEdge>	Kolejka priorytetowa krawędzi incydentnych do danego wierzchołka
weight	long	Suma wag krawędzi MST
vertices	PrimVertex[]	Listy sąsiedztw – reprezentacja grafu
Prim (PrimVertex[])	Prim	Konstruktor klasy Prim, inicjalizujący graf.
Prim	klasa	Klasa realizująca algorytm Prima
primMST	void	Metoda realizująca kroki algorytmu Prima
rand	Rand	Zmienna pomocnicza – element potrzebny do losowego wyboru wierzchołka startowego
randV	int	Wierzchołek startowy, zmienna pomocnicza
offer(PrimEdge)	void	Metoda realizująca dodawanie krawędzi do kolejki priorytetowej
isEmpty()	boolean	Metoda zwracająca *true*, gdy kolejka priorytetowa jest pusta, w przeciwnym wypadku *false*
push(PrimEdge)	void	Metoda dodająca krawędź na stos *minSpanningTree*

rys. 3.13. Składowe implementacji algorytmu Prima

Składowe pomocnicze implementacji algorytmu Prima – rys. 3.13

Składowe pomocnicze		
Składowa	Rodzaj	Opis
time, timeStart, timeEnd	long	Zmienne pomocnicze do pomiaru czasu działania algorytmu
PrimEdge	PrimEdge	Klasa stanowiąca model krawędzi grafu – posiada koszt, wierzchołek początkowy i końcowy. Ważna jest tu metoda compareTo(), dzięki której porównywany atrybut podczas dodawania elementu do kolejki priorytetowej jest zdefiniowany
showMST(Deque<PrimEdge>)	String	Metoda odpowiedzialna za wygenerowanie drzewa MST w postaci tekstu do wyświetlenia w konsoli.
PrimVertex	klasa	Model reprezentujący wierzchołek w postaci jego listy sąsiedztwa

rys. 3.14. Składowe pomocnicze implementacji algorytmu Prima

Implementacja algorytmu Prima – rys.3.15

```java
import java.util.*;
import java.util.concurrent.TimeUnit;

public class Prim {
    private boolean[] visitedVertices;
    private Deque<PrimEdge> minSpanningTree;
    private PriorityQueue<PrimEdge> adjacencyEdges;
    private long weight;
    private long time;
    private PrimVertex[] vertices;


    public Prim(PrimVertex[] vertices) {
        this.vertices = vertices;
    }

    public void primMST(){
        weight = 0;
        visitedVertices = new boolean[vertices.length];
        minSpanningTree = new ArrayDeque<>();
        adjacencyEdges = new PriorityQueue<>(vertices.length);
        long timeStart = TimeUnit.MILLISECONDS.toMicros(System.currentTimeMillis());

        Random rand = new Random();
        int randV = rand.nextInt(vertices.length - 1);

        visitedVertices[randV] = true;

        for (PrimEdge edge : vertices[randV].neighbors) {
            adjacencyEdges.offer(edge);
        }

        while (!adjacencyEdges.isEmpty()) {
            PrimEdge edge = adjacencyEdges.remove();

            if (visitedVertices[edge.end]) {
                continue;
            }

            visitedVertices[edge.end] = true;
            minSpanningTree.push(edge);
            weight+=edge.cost;

            for (PrimEdge visitedEdge : vertices[edge.end].neighbors) {
                adjacencyEdges.offer(visitedEdge);
            }
        }

        long timeStop = TimeUnit.MILLISECONDS.toMicros(System.currentTimeMillis());

        time += (timeStop-timeStart);
    }
```

rys. 3.15. Implementacja algorytmu Prima

Implementacja algorytmu Prima c.d.– rys.3.16

```
54      public void showTime(int iterations){
55          System.out.println("CZAS WYKONYWANIA ALGORYTMU PRIMA WYNIÓSŁ: "+ (time/iterations) + " mikrosekund");
56          System.out.println("SUMA WAG: " + weight);
57          System.out.println("Na MST składają się nastepujące krawędzie: ");
58          System.out.println(showMST(minSpanningTree));
59      }
60
61      private String showMST(Deque<PrimEdge> tree){
62          StringBuilder builder = new StringBuilder();
63          while(!tree.isEmpty()){
64              PrimEdge edge = tree.pop();
65              builder.append(edge.start).append(" -> ")
66                      .append(edge.end).append(": ")
67                      .append(edge.cost)
68                      .append("\n");
69          }
70
71          return builder.toString();
72      }
73
74  }
```

rys. 3.16. Implementacja algorytmu Prima c.d.

Implementacja klas pomocniczych dla algorytmu Prima – rys.3.17

```
1   public class PrimEdge implements Comparable<PrimEdge> {
2       int end;
3       int cost;
4
5       public PrimEdge(int end, int cost) {
6           this.end = end;
7           this.cost = cost;
8       }
9
10      public int compareTo(PrimEdge edge) {
11          return this.cost - edge.cost;
12      }
13  }
```

rys. 3.17. Implementacja klas pomocniczych dla algorytmu Prima

Implementacja klas pomocniczych dla algorytmu Prima c.d. – rys.3.18

```java
import java.util.ArrayList;

public class PrimVertex {
    ArrayList<PrimEdge> neighbors;

    public PrimVertex() {
        this.neighbors = new ArrayList<>();
    }
}
```

rys. 3.18. Implementacja klas pomocniczych dla algorytmu Prima c.d.

3. Kod źródłowy algorytmu Kruskala

Implementacja struktury zbiorów rozłącznych wykorzystuje techniki kompresji ścieżek oraz scalania po randze. Pozwala to na bardziej efektywne wykonywanie operacji niezbędnych do realizacji algorytmu. Kompresja ścieżek polega na „spłaszczeniu" drzewa, co skraca czas dostępu do poszczególnych elementów. Scalanie po randze natomiast powoduje, że płytsze drzewo jest zawsze dołączane do korzenia głębszego, co pozwala ograniczyć zwiększanie się głębokości drzewa.
Składowe implementacji algorytmu Kruskala – rys. 3.19:

Składowa	Rodzaj	Opis
V	int	Liczba wierzchołków grafie
E	int	Liczba krawędzi w grafie
edge	EdgeK[]	Tablica zawierająca wszystkie krawędzie w grafie
result	EdgeK[]	Tablica zawierająca krawędzie tworzące MST
e, i, x, y	int	Zmienne pomocnicze
subsets	Subset[]	Tablica zawierająca zbiory w strukturze zbiorów rozłącznych
next_edge	EdgeK	Następna krawędź o najmniejszej wadze, która może zostać dodana do MST
KruskalAlgorithm	klasa	Klasa realizująca algorytm Kruskala
EdgeK	klasa	Klasa realizująca krawędź w implementacji algorytmu Kruskala
subset	klasa	Klasa realizująca podzbiór w implementacji zbiorów rozłącznych
find()	metoda	Metoda realizująca znajdywanie w implementacji zbiorów rozłącznych
union()	metoda	Metoda realizująca scalanie w implementacji zbiorów rozłącznych
KruskalMST	metoda	Metoda realizująca algorytm Kruskala

rys. 3.19. Składowe implementacji algorytmu Kruskala

Listing kodu źródłowego implementacji algorytmu Kruskala:

```
import java.util.*;
import java.lang.*;

public class KruskalAlgorithm {

        class EdgeK implements Comparable<EdgeK>
        {
            int source, destination, weight;
            public int compareTo(EdgeK compareEdge)
            {
                return this.weight - compareEdge.weight;
            }
        }

        class subset
        {
            int parent, rank;
        };

        int V, E;
        EdgeK edge[];

        KruskalAlgorithm(int v, int e)
        {
            V = v;
            E = e;
            edge = new EdgeK[E];
            for (int i=0; i<e; ++i)
                edge[i] = new EdgeK();
        }

        // znajduje subset z danym wierzchołkiem
        int find(subset subsets[], int i)
        {
            if (subsets[i].parent != i)
                subsets[i].parent = find(subsets, subsets[i].parent);

            return subsets[i].parent;
        }

        // realizuje scalanie zbiorów
        void Union(subset subsets[], int x, int y)
        {
            int xroot = find(subsets, x);
            int yroot = find(subsets, y);

            if (subsets[xroot].rank < subsets[yroot].rank)
                subsets[xroot].parent = yroot;
            else if (subsets[xroot].rank > subsets[yroot].rank)
                subsets[yroot].parent = xroot;

            else
            {
                subsets[yroot].parent = xroot;
                subsets[xroot].rank++; }}
```

rys. 3.20. Implementacja algorytmu Kruskala

Listing kodu źródłowego implementacji algorytmu Kruskala c.d.:

```
        // realizuje algorytm Kruskala
        void KruskalMST()
        {
            EdgeK result[] = new EdgeK[V];  // wynik
            int e = 0;
            int i = 0;
            for (i=0; i<V; ++i)
                result[i] = new EdgeK();

            // Sortuje krawedzie niamalejąco
            Arrays.sort(edge);

            subset subsets[] = new subset[V];
            for(i=0; i<V; ++i)
                subsets[i]=new subset();

            for (int v = 0; v < V; ++v)
            {
                subsets[v].parent = v;
                subsets[v].rank = 0;
            }

            i = 0;

            while (e < V - 1)
            {
                // Wybiera krawędź o najmniejszej wadze
                EdgeK next_edge = new EdgeK();
                next_edge = edge[i++];

                int x = find(subsets, next_edge.source);
                int y = find(subsets, next_edge.destination);

                // Jeżeli kawedz nie tworzy cklu dodaje ja do rozwiazania
                if (x != y)
                {
                    result[e++] = next_edge;
                    Union(subsets, x, y);
                }
            }

            // print the contents of result[] to display
            // the built MST
            System.out.println("Na MST skladaja się nastepujace
krawedzie:");
            for (i = 0; i < e; ++i)
                System.out.println(result[i].source +" -- " +
                        result[i].destination +" == " + result[i].weight);
        }
}
```

rys. 3.21. Implementacja algorytmu Kruskala c.d.

Spis rysunków

Bibliografia

[1] *http://www.algorytm.org/algorytmy-grafowe/algorytm-prima.html*

[2] *http://algorytmika.wikidot.com/mst*

[3] Jacek Wojciechowski, Krzysztof Pieńkosz: *Grafy i sieci*, Wydawnictwo Naukowe PWN SA, Warszawa 2013

[4] *http://www.ams.org/proc/1956-007-01/S0002-9939-1956-0078686-7/*