Politechnika Wrocławska

Wydział Informatyki i Telekomunikacji

Projektowanie efektywnych algorytmów

Algorytmy Dokładne

Etap nr 1

Prowadzący:

dr inż. Jarosław Rudy

Grupa:

A - Wtorek 17:05 TP

Autor:

Daniel Glazer, 252743

Wrocław 16.11.2021

Spis treści

Spis rysunków

1 Wprowadzenie teoretyczne

1.1 Opis problemu

Problemem, który rozważany jest w tej pracy to problem komiwojażera. Problem komiwojażera sprowadza się do znalezienia w zadanym grafie pełnym cyklu Hamiltona. Cykl ten jest drogą zamkniętą, czyli taką drogą, w której każdy wierzchołek grafu został odwiedzony jeden raz oprócz wierzchołka początkowego, w którym zaczyna i kończy się nasza droga. Można rozróżnić dwa rodzaje wyżej wymienionego problemu: wariant symetryczny oraz asymetryczny. W przypadku pierwszego z nich odległość pomiędzy wierzchołkami X i Y jest taka sama zarówno podczas przejścia z wierzchołka X do Y jak i z Y do X. Natomiast w drugim wariancie koszt podczas przejścia może być różny.

1.2 Opis metody

1.2.1 Przegląd zupełny

Metoda przeglądu zupełnego polega na znalezieniu wszystkich dopuszczalnych rozwiązań problemu oraz wybraniu ekstremum, które w tej sytuacji będzie minimum. Efektem działania algorytmu jest uzyskanie rozwiązania problemu, które jest jednocześnie rozwiązaniem optymalnym. Implementacja tego algorytmu zalicza się do kategorii tych prostszych, ale samo sprawdzenie wszystkich możliwych kombinacji jest bardzo czasochłonnym rozwiązaniem i dla większych instancji wręcz nieakceptowalnym. Widoczne jest to szczególnie w problemach o dużej złożoności takich jak np. rozwiązywany w aktualnym zadaniu problem komiwojażera, którego złożoność wynosi $O(n!)$. Przeszukiwanie metodą siłową ma też swoje zalety, bo jednak posiada pewność znalezienia rozwiązania (oczywiście tylko wtedy jeśli istnieje jakieś rozwiązanie) oraz mamy pewność, że to rozwiązanie jest optymalne.

1.2.2 Algorytm podziału i ograniczeń

Metoda podziału i ograniczeń służy do rozwiązywania problemów optymalizacyjnych. Opiera się na przeszukiwaniu drzewa stanów, które posiada możliwe ścieżki jakimi może podążać algorytm w celu znalezienia rozwiązania optymalnego. Algorytm zaczyna w korzeniu, który w przypadku problemu komiwojażera jest wierzchołkiem startowym. Następnie przy pomocy usuwania wiersz i kolumn z macierzy, ogranicza macierz dla każdego węzła wychodzącego z naszego węzła badanego. Na podstawie tych macierzy oblicza koszt uzyskania kolejnego węzła. Na podstawie wartości wszystkich obliczonych i nieodwiedzonych węzłów wybiera ten, gdzie wartość jest najlepsza dla rozwiązania, w przypadku komiwojażera pożądana jest wartość najmniejsza. Algorytm kończy swoje działanie w momencie, w którym od korzenia do liścia posiadamy tyle poziomów ile wierzchołków ma zadany graf pełny. Dla problemu komiwojażera wykorzystując tą metodę złożoność czasowa wynosi $O(n^2 2^n)$.

1.2.3 Programowanie dynamiczne

Metoda programowania dynamicznego opiera się na podziale zadanego problemu, który w naszym przypadku jest problemem komiwojażera na podproblemy. Podczas działania algorytmu wyznaczane są wartości minimalne dla poszczególnych zbiorów zaczynając od zbiorów jednoelementowych, dla których wartością minimalna jest długość ścieżki pomiędzy wierzchołkiem ze zbioru, a wierzchołkiem początkowym. Przy kolejnych etapach algorytm wykorzystuje do obliczenia długości ścieżki dla zbioru posiadającego więcej niż jeden element, obliczone koszty uzyskania określonej ścieżki ze zbiorów o jeden element mniejszy. Wyniki obliczeń powinny być zapisywane i w przypadku powtarzania się zbiorów powinny być wykorzystywane przy obliczaniu kolejnych długości. Po obliczeniu minimum dla zbioru posiadającego wszystkie wierzchołki algorytm zwraca optymalne rozwiązanie. Dla takiego algorytmu złożoność obliczeniowa wynosi $O(n^2 2^n)$, a złożoność pamięciowa $O(n2^n)$.

2 Opis implementacji algorytmów

2.1 Wstęp

Na starcie każdego wywoływanego algorytmu, wywoływana jest klasa *Matrix*, której celem jest sczytanie danych z pliku i utworzenie dwuwymiarowej tablicy dynamicznej posiadająca długości ścieżek. W przypadku testów zamiast sczytywania danych z pliku, został generowany pseudolosowy graf pełny o określonym rozmiarze i rodzaju,

2.2 Przegląd zupełny

Na początku algorytmu tworzony jest wektor - *nodes*, który zawiera wszystkie wierzchołki oprócz startowego. Następnie w pętli, która się wykonuje do momentu aż zostaną sprawdzane wszystkie permutacje. Kolejne permutacje są wyznaczane za pomocą metody *next_permutaion* z biblioteki *algorithm*, do której przekazywany jest wektor *nodes*. Dla każdej permutacji obliczany jest koszt ścieżki oraz zapisywany do tablicy ciąg przebytych wierzchołków. W momencie, w którym długość obliczonej ścieżki jest mniejsza niż dotychczasowa minimalna ścieżka, kopiujemy obecną długość cyklu do najkrótszej ścieżki oraz kopiujemy tablicę zawierająca przebyte wierzchołki. Złożoność czasowa dla algorytmu wynosi $O(n! \cdot n)$ ponieważ sprawdzamy wszystkie permutacje, których jest $n!$ oraz dla każdej permutacji jest obliczany koszt ścieżki. Kosz ścieżki jest liczony w pętli, która wykonuje się n razy. Na złożoność pamięciową składają się 2 tablice o długości $n+1$ oraz wektor o długości n, więc złożoność pamięciowa wynosi $O(n)$.

2.3 Algorytm podziału i ograniczeń

Algorytm podziału i ograniczeń rozpoczyna się od utworzenia węzła początkowego, który posiada macierz wag, informacje o poziomie i wektor odpowiadający za przechowywanie cyklu, na którym obecnie znajduje się wierzchołek początkowy. Następnie macierz zawierająca się w węźle zostaje zredukowana poprzez odjęcie najmniejszego elementu z wiersza i kolumny. Odjęte wartości są dodawane do kosztu węzła. Taki węzeł zostaje dodany do kolejki priorytetowej, która została zaimplementowana w taki sposób aby na szczycie znajdował

się węzeł o najmniejszym koszcie. Następnie wywoływana jest metoda, która posiada pętlę odpowiedzialną za liczenie kosztów dla węzłów wychodzących z węzła, który znajduje się na szczycie kolejki. Dla każdego wychodzącego węzła na podstawie macierzy rodzica, ustawiane są nieskończoności w miejscach wiersza rodzica i kolumny dziecka oraz zostaje dodany kosztu dostępu z wierzchołka rodzica do wierzchołka dziecka. Następnie jest obliczany koszt dla wychodzącego węzła na podstawie zredukowanej macierzy, który jest dodawany do wcześniej uzyskanego kosztu dostępu. Każdy potomek jest dodawany do kolejki. W momencie, w którym dodamy wszystkich potomków dla danego węzła. Uruchamiamy kolejną iterację pętli dla węzła ze szczytu kolejki. Pętla wykonuje się do momentu, aż kolejka będzie pusta lub znajdziemy węzeł, którego poziom jest równy rozmiarowi macierzy minus jeden. Czasowa złożoność dla tego algorytmu wynosi $O((n-1)! \cdot n^2)$ ponieważ w najgorszym przypadku sprawdzane są wszystkie węzły, których jest $(n-1)!$ oraz dla każdego węzła jest redukowana macierz w podwójnej pętli co daje n^2. Na złożoność pamięciową składa się $(n-1)!$ węzłów z których każdy posiada tablice o rozmiarze n^2 oraz wektor o długości n odpowiedzialny za kolejność wierzchołków ścieżki, więc ostatecznie złożoność pamięciowa wynosi $O(n^2(n-1)!)$.

2.4 Programowanie dynamiczne

Algorytm rozpoczyna swoją pracę od utworzenia wektora, który jest zbiorem zawierającym wszystkie wierzchołki oprócz początkowego. Na podstawie tego wektora, tworzony jest węzeł początkowy. Następnie wywoływana jest metoda *findMinimum* dla węzła początkowego. W tej metodzie znajduje się pętla odpowiedzialna za tworzenie węzłów potomków, którzy zawierają kolejne zmniejszone zbiory oraz koszty uzyskania tych potomków. Dla każdego potomka wywoływana jest metoda *findMinimum* do momentu, aż wektor zbioru jest pusty, a co za tym idzie nasz węzeł prowadzi tylko do wierzchołka startowego. Następnie szukane jest minimum dla węzła rodzica na podstawie kosztów uzyskanych przez jego potomków. Algorytm wykonuje się do momentu, aż dla węzła początkowego zostanie znalezione minimum. Złożoność czasowa dla algorytmu wynosi $O(2^n \cdot n^2)$ ponieważ sprawdzamy wszystkie permutacje zbiorów, których jest 2^n, tworzymy pomniejszony wektor z którego usuwamy numer poszczególnego węzła, więc musimy przesunąć wskaźnik wektora n razy oraz dla każdego zbioru obliczane jest minimum n razy. Na złożoność pamięciową składają się wektory zbiorów o długości n dla wszystkich permutacji zbiorów, których jest 2^n, więc złożoność pamięciowa wynosi $O(2^n \cdot n)$.

3 Testy zaimplementowanych algorytmów

3.1 Poprawność badanych algorytmów

```
Otworzono plik mó.atsp
Wczytano dane
158   97.5%
148   85%
144   80%
142   77.5%
116   45%
80   0%
0->5->1->2->3->4->0
```

(a) Przegląd zupełny

```
Otworzono plik mó.atsp
Wczytano dane
72   -10%
76   -5%
80   0%
0->5->1->2->3->4->0
```

(b) Algorytm podziału i ograniczeń

```
Otworzono plik mó.atsp
Wczytano dane
80   0%
0->5->1->2->3->4->0
```

(c) Programowanie dynamiczne

Rysunek 1: Badanie poprawności algorytmów dla pliku m6.atsp

```
Otworzono plik m10.atsp
Wczytano dane
499   135.377%
444   109.434%
433   104.245%
383   80.6604%
364   71.6981%
362   70.7547%
352   66.0377%
347   63.6792%
345   62.7359%
323   52.3585%
304   43.3962%
302   42.4528%
301   41.9811%
294   38.6792%
287   35.3774%
278   31.1321%
274   29.2453%
273   28.7736%
272   28.3019%
257   21.2264%
250   17.9245%
248   16.9811%
239   12.7358%
234   10.3774%
219   3.30189%
213   0.471698%
212   0%
0->3->4->2->8->7->6->9->1->5->0
```

(a) Przegląd zupełny

```
Otworzono plik m10.atsp
Wczytano dane
170   -19.8113%
189   -10.8491%
190   -10.3774%
192   -9.43396%
196   -7.54717%
198   -6.60377%
201   -5.18868%
203   -4.24528%
204   -3.77358%
205   -3.30189%
206   -2.83019%
207   -2.35849%
208   -1.88679%
209   -1.41509%
210   -0.943396%
211   -0.471698%
212   0%
0->5->1->9->6->7->8->2->4->3->0
```

(b) Algorytm podziału i ograniczeń

```
Otworzono plik m10.atsp
Wczytano dane
212   0%
0->3->4->2->8->7->6->9->1->5->0
```

(c) Programowanie dynamiczne

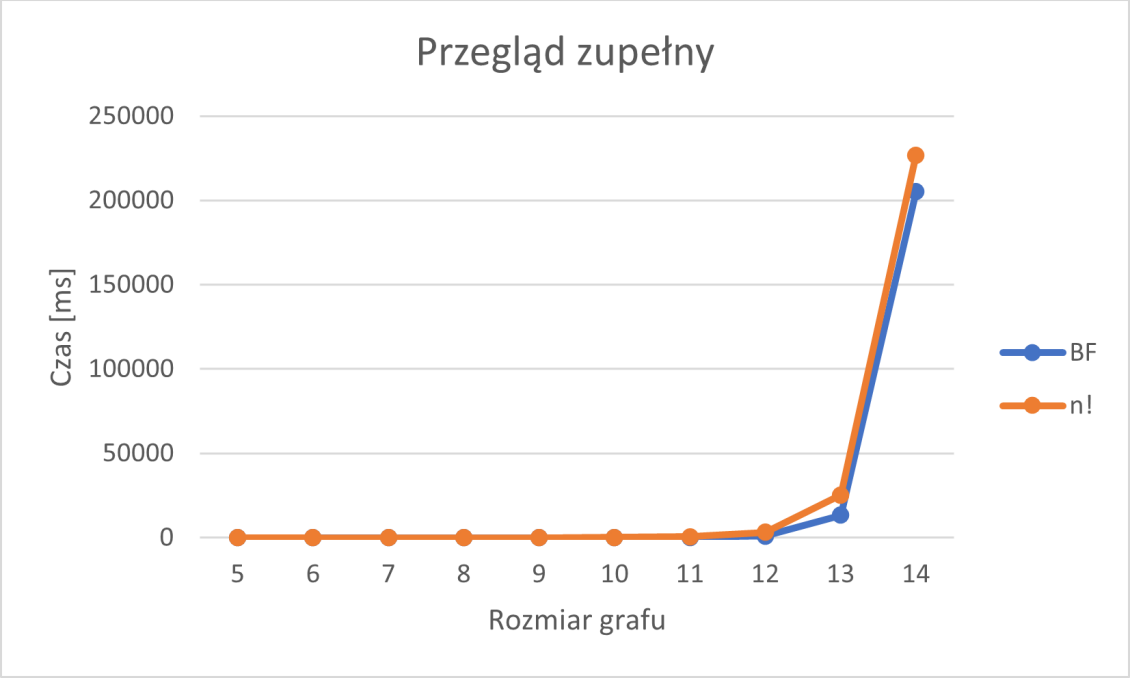
Rysunek 2: Badanie poprawności algorytmów dla pliku m10.atsp

3.2 Wyniki pomiarów

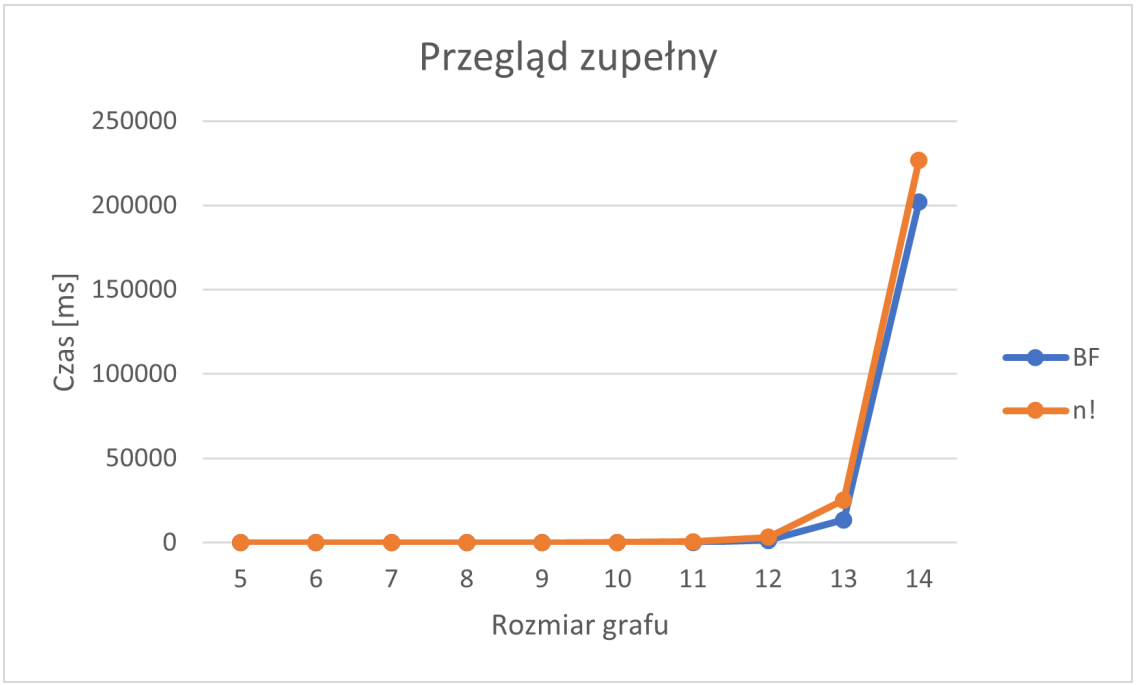
Liczba wierzchołków	Czas [ms]					
	BF	BB	DP	BF	BB	DP
	Nieskierowany			Skierowany		
5	0,0027	0,11	0,12	0,0020	0,024	0,024
6	0,01	0,10	0,20	0,0040	0,039	0,077
7	0,04	0,74	0,85	0,45	0,11	0,69
8	0,16	0,47	6,39	0,14	0,21	4,26
9	1,13	0,90	31,52	2,27	1,05	22,80
10	8,61	0,65	313,83	10,22	0,40	265,14
11	92,13	3,21	2220,78	129,58	0,33	1840,38
12	1058,15	8,24	17899,60	1284,79	0,95	19836,20
13	13361,50	11,30	-	13548,40	0,69	-
14	205183,00	22,36	-	202000,00	0,59	-

Tabela 1: Pomiary czasu wykonywanie się poszczególnych algorytmów: BF - przegląd zupełny, BB - metoda podziału i ograniczenia, DP - dynamiczne programowanie dla określonej liczby wierzchołków

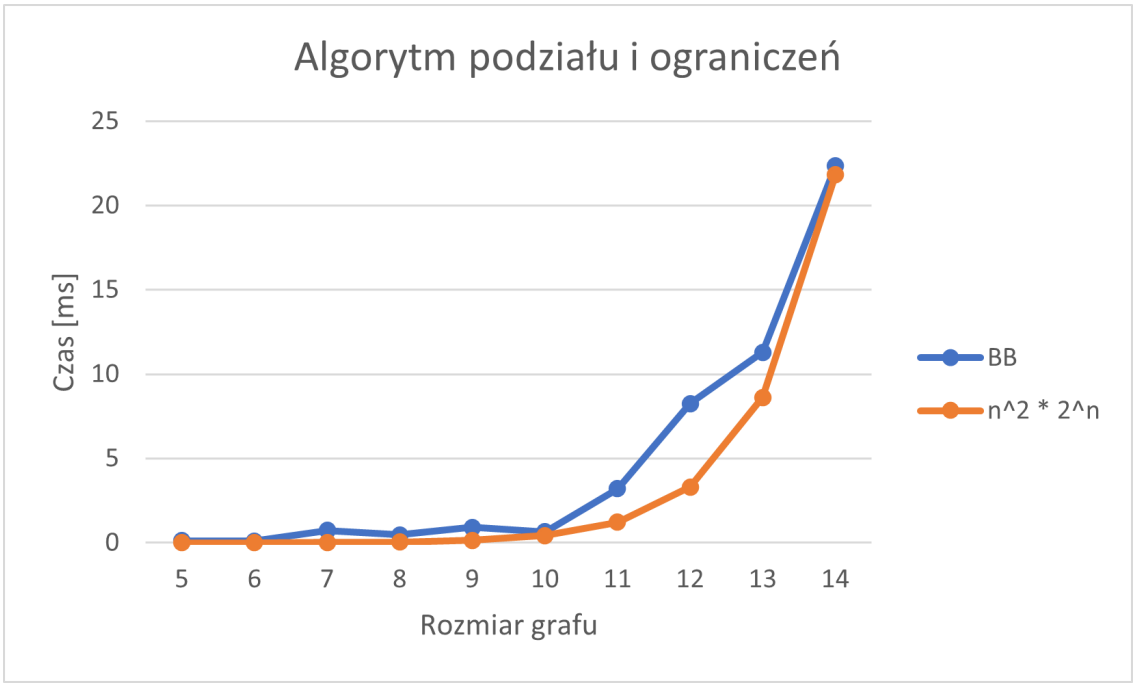
3.3 Wykresy złożoności



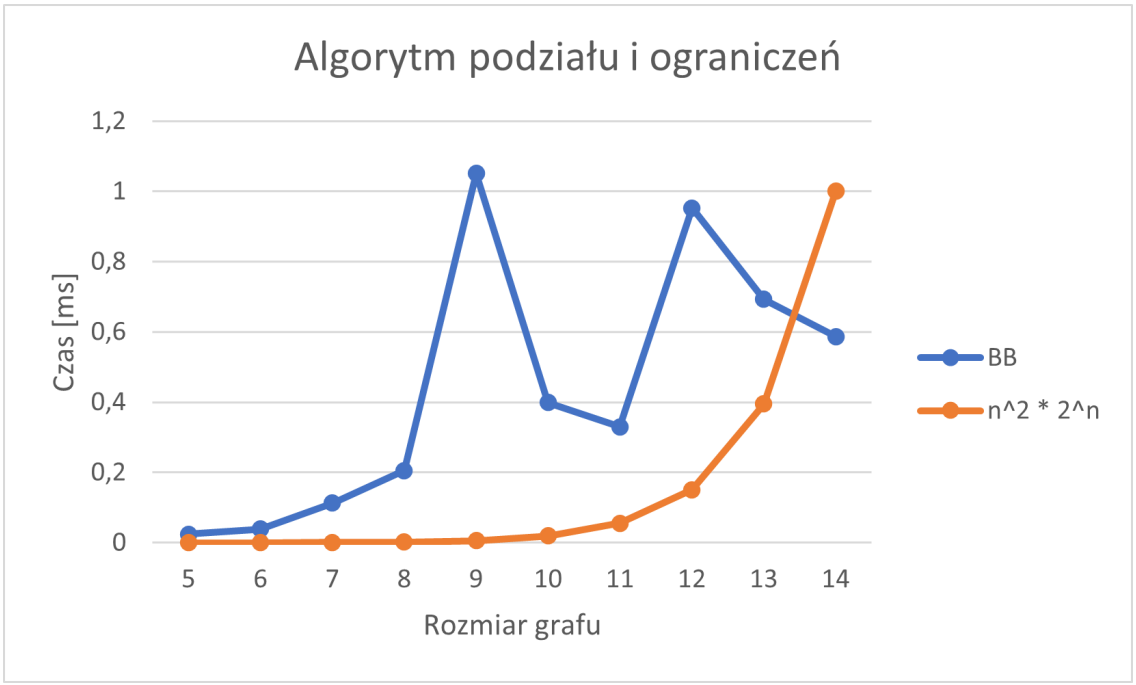
Rysunek 3: Porównanie złożoności przeglądu zupełnego z jego wartością teoretyczną dla grafu nieskierowanego



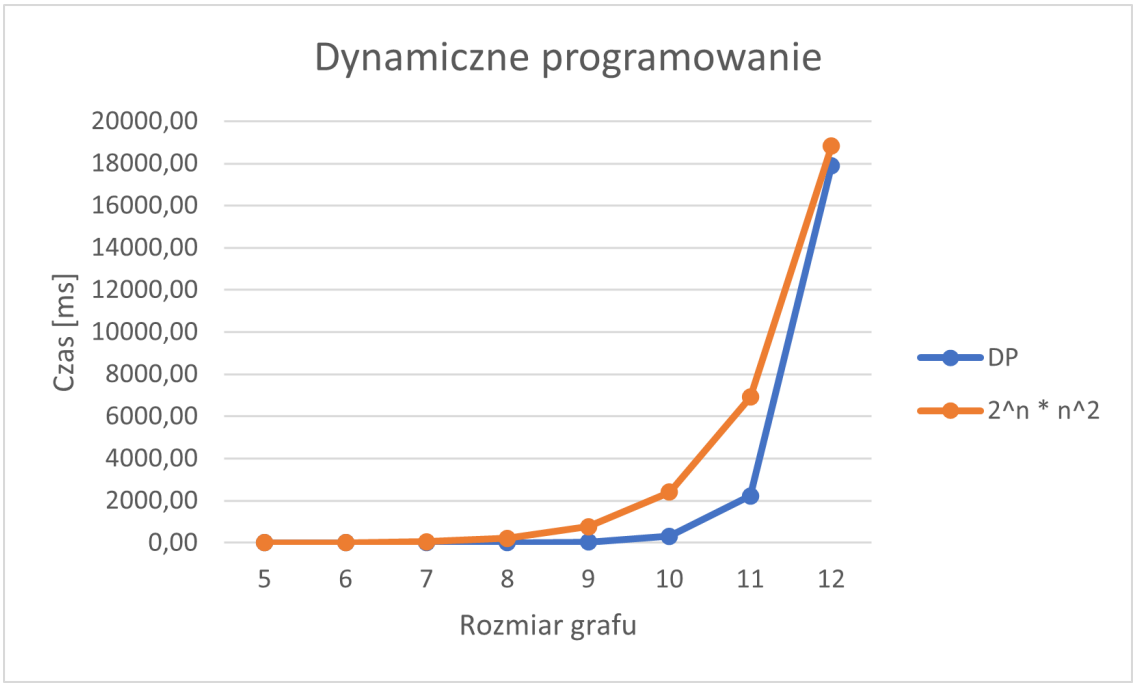
Rysunek 4: Porównanie złożoności przeglądu zupełnego z jego wartością teoretyczną dla grafu skierowanego



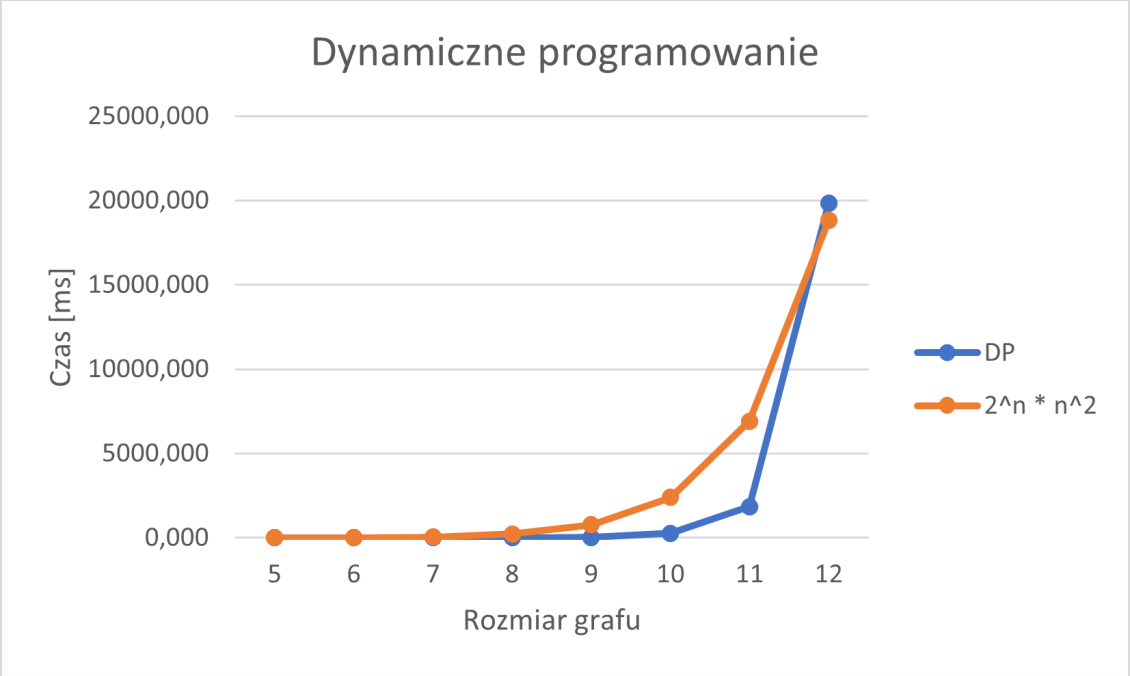
Rysunek 5: Porównanie złożoności algorytmu podziału i ograniczeń z jego wartością teoretyczną dla grafu nieskierowanego



Rysunek 6: Porównanie złożoności algorytmu podziału i ograniczeń z jego wartością teoretyczną dla grafu skierowanego

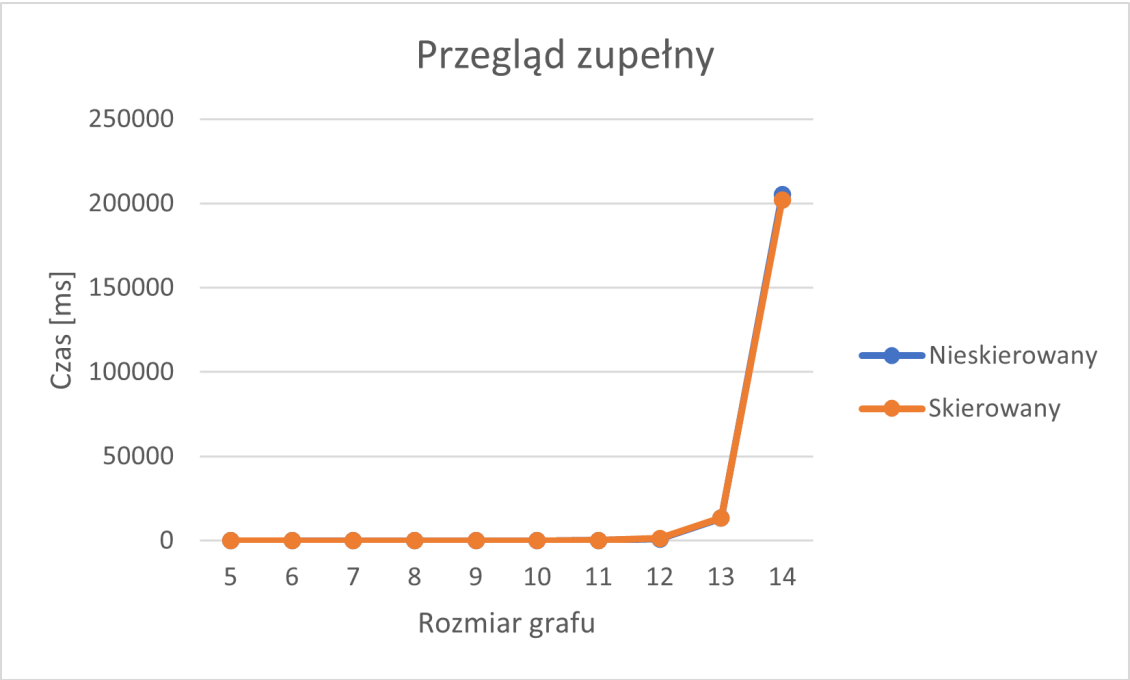


Rysunek 7: Porównanie złożoności programowania dynamicznego z jego wartością teoretyczną dla grafu nieskierowanego

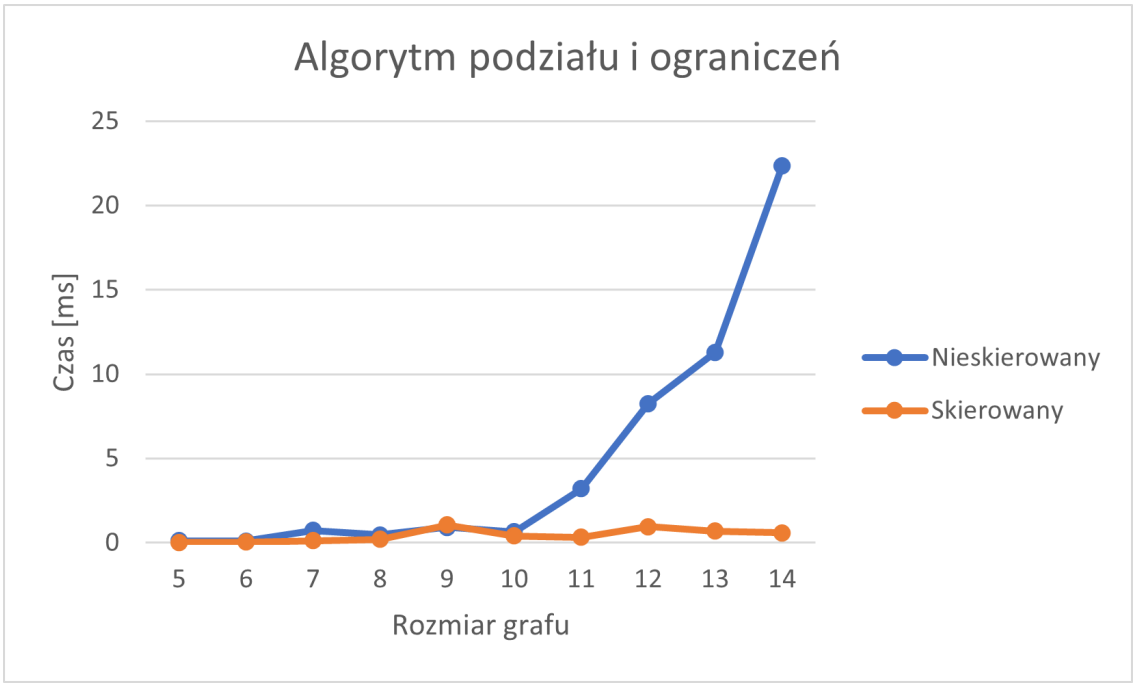


Rysunek 8: Porównanie złożoności programowania dynamicznego z jego wartością teoretyczną dla grafu skierowanego

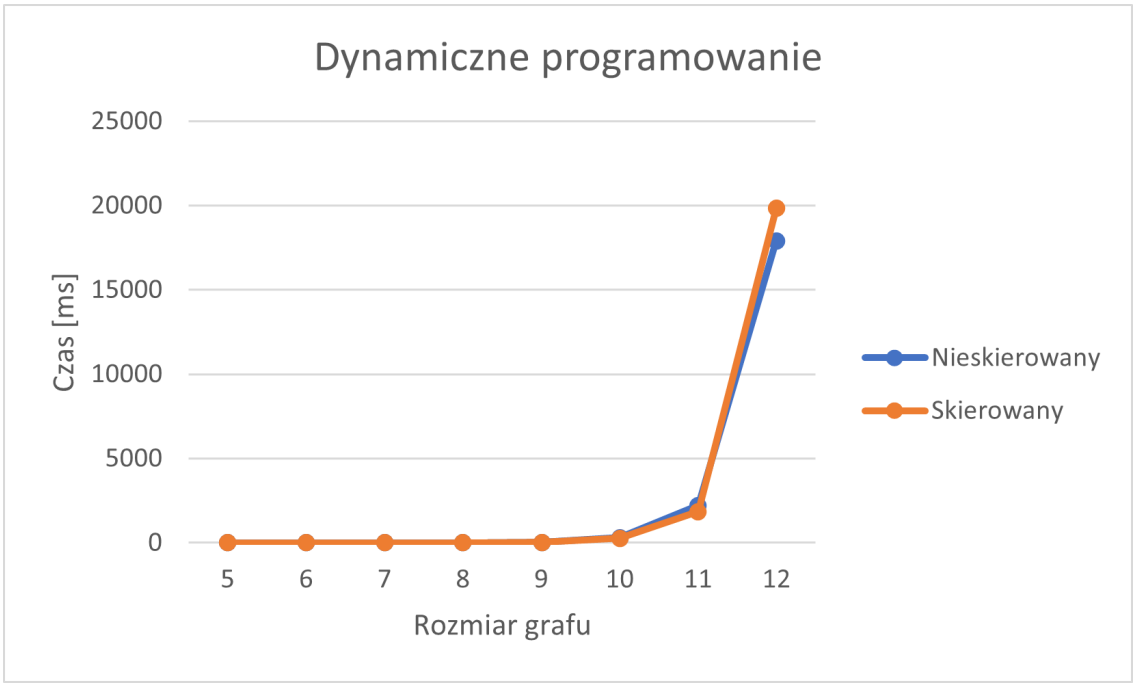
3.4 Wykresy porównujące wpływ rodzaju grafu na złożoność



Rysunek 9: Porównanie złożoności przeglądu zupełnego dla różnych rodzajów grafów

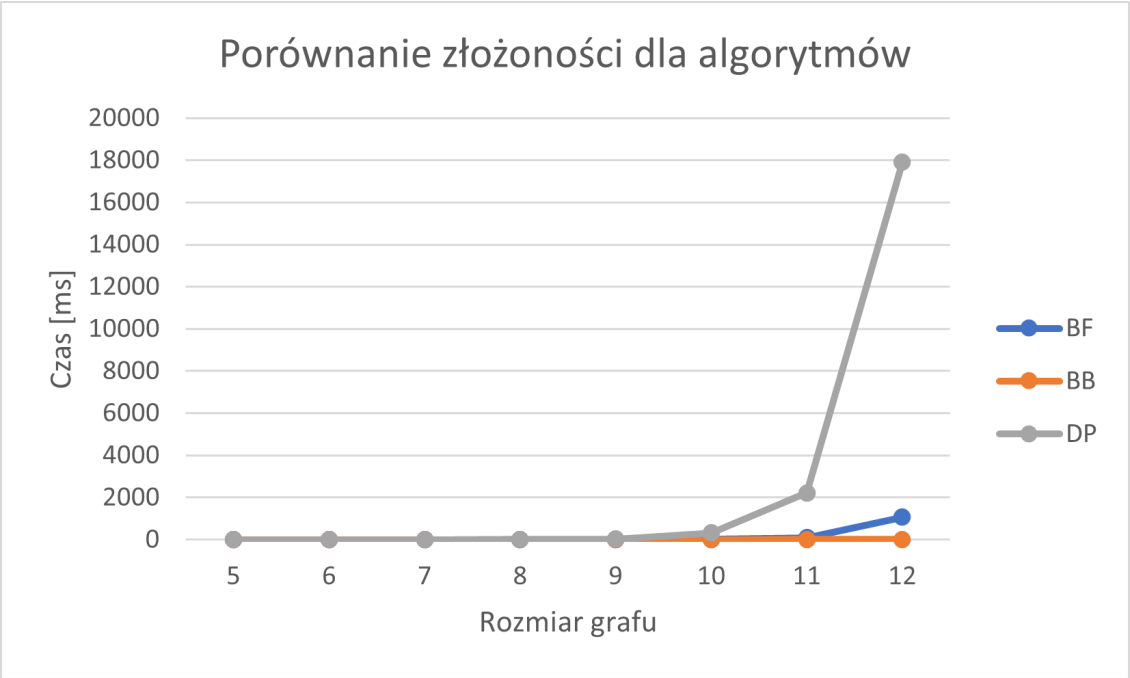


Rysunek 10: Porównanie złożoności algorytmu podziału i ograniczeń dla różnych rodzajów grafów



Rysunek 11: Porównanie złożoności programowania dynamicznego dla różnych rodzajów grafów

3.5 Wykres porównania złożoności dla wszystkich badanych algorytmów



Rysunek 12: Porównanie złożoności programowania dynamicznego dla różnych rodzajów grafów

4 Wnioski

4.1 Przegląd zupełny

Z wyników przedstawionych na wykresach złożoności można stwierdzić, że złożoność algorytmu przeglądu zupełnego jest porównywalna z wartością teoretyczną wynoszącą $O(n!)$ jest to zauważalne dla obydwóch rodzajów grafu. Można również zauważyć, że rodzaj grafu nie wpływa na czas wykonywania się algorytmu.

4.2 Algorytm podziału i ograniczeń

Dla algorytmu *B&B* sytuacja wygląda trochę inaczej niż w przypadku przeglądu zupełnego. Z wykresu złożoności wynika, że dla grafu nieskierowanego złożoność algorytmu jest zbliżona do zakładanego $O(n^2 \cdot 2^n)$, trochę inaczej sprawa ma się w przypadku grafu skierowanego. Wykres dla grafu skierowanego posiada piki i spadki, z których trudno wywnioskować złożoność. Taki wygląd wykresu może być spowodowany nieidealnym generatorem liczb pseudolosowych, który powoduje, że dla niektórych mniejszych instancji mamy przypadek, w którym musimy sprawdzić wszystkie węzły, a dla większych instancji mamy przypadek, w którym ilość węzłów jest zbliżona albo równa ilości wierzchołków. Ale zauważalne jest na pewno to, że dla grafu skierowanego średnie wyniki pomiarów są mniejsze niż dla grafu nieskierowanego. Wynika to najprawdopodobniej z sposobu odejmowania wierszy i kolumn z macierzy, bo pozwala nam na częstsze redukowanie macierzy.

4.3 Programowanie dynamiczne

Analizując złożoność czasową dla programowania dynamicznego można zauważyć, że nie jest zbliżona do wartości teoretycznej wynoszącej $(n^2 \cdot 2^n)$, a raczej do złożoności $O(n!)$. Wynika to z faktu, że przy obliczaniu kosztów dla poszczególnych zbiorów nie są one wykorzystywane w przypadku, w którym dany zbiór się powtórzy. A poprzez obliczanie wartości dla każdego zbioru, ten algorytm bardziej przypomina przegląd zupełny dlatego też bierze się złożoność $O(n!)$. Rodzaj grafu nie wpływa znacząco na otrzymywany wynik pomiaru.

4.4 Ogólne

Porównując wszystkie wyniki uzyskane podczas pomiarów można stwierdzić, że najbardziej wydajnym algorytmem jest algorytm *B&B*, ponieważ stara się szukać złotego środka pomiędzy złożonością pamięciową, a złożonością czasową. Kolejnym algorytmem, który według złożoności czasowej powinien znaleźć się na drugim miejscu jest programowanie dynamiczne, ale z powodu błędnej implementacji bardziej przypomina przegląd zupełny. Z powodu operacji, które wykonywane są na węzłach i zbiorach staje się zdecydowanie mniej wydajny niż *BF* mimo, że oby dwa algorytmy w obecnym stanie posiadają podobną złożoność czasową. Przegląd zupełny pomimo, że zapewnia znalezienie zawsze optymalnego rozwiązania jest jednak bardzo czasochłonnym sposobem znalezienia rozwiązania co skreśla go dla większych instancji.