Sprawozdanie

Projektowanie Efektywnych Algorytmów

Zadanie projektowe nr 1

Problem komiwojażera dla programowania dynamicznego i metody brute-force

Autor: **Kamil Kamyszek**

Prowadzący: **dr inż. Jarosław Mierzwa**

Data: 13.11.2018r.

1 Wstęp

1.1 Opis zadania projektowego

Należy zaimplementować algorytm brute force i programowania dynamicznego dla problemu komiwojażera oraz dokonać testów polegających na pomiarze czasu działania algorytmu w zależności od wielkości.

1.2 Wstęp teoretyczny dla programowania dynamicznego

1.2.1 Problem komiwojażera

Problem komiwojażera jest zagadnieniem optymalizacyjnym, polegającym na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Nazwa pochodzi od typowej ilustracji problemu, przedstawiającej go z punktu widzenia wędrownego sprzedawcy (komiwojażera): dane jest n miast, które komiwojażer ma odwiedzić, oraz odległość / cena podróży / czas podróży pomiędzy każdą parą miast. Celem jest znalezienie najkrótszej / najtańszej / najszybszej drogi łączącej wszystkie miasta, zaczynającej się i kończącej się w określonym punkcie.

1.2.2 Programowanie dynamiczne

Programowanie dynamiczne jest techniką projektowania algorytmów, stosowaną głównie do rozwiązywania zagadnień optymalizacyjnych. Jest alternatywą dla niektórych zagadnień rozwiązywanych za pomocą algorytmów zachłannych. Programowanie dynamiczne opiera się na podziale rozwiązywanego problemu na podproblemy. Zagadnienia odpowiednie dla programowania dynamicznego cechuje to, że zastosowanie do nich metody brute-force prowadzi do ponadwielomianowej liczby rozwiązań podproblemów, podczas gdy sama liczba różnych podproblemów jest wielomianowa. Klucz do zaprojektowania algorytmu tą techniką leży w znalezieniu równania rekurencyjnego opisującego optymalną wartość funkcji celu dla danego problemu jako funkcji optymalnych wartości funkcji celu dla podproblemów o mniejszych rozmiarach. Programowanie dynamiczne znajduje optymalną wartość funkcji celu dla całego zagadnienia, rozwiązując podproblemy od najmniejszego do największego i zapisując optymalne wartości w tablicy. Pozwala to zastąpić wywołania rekurencyjne odwołaniami do odpowiednich komórek wspomnianej tablicy i gwarantuje, że każdy podproblem jest rozwiązywany tylko raz. Rozwiązanie ostatniego z rozpatrywanych podproblemów jest na ogół wartością rozwiązania zadanego zagadnienia. Programowanie dynamiczne jest jedną z bardziej skutecznych technik rozwiązywania problemów NP-trudnych. Problem komiwojażera (TSP) jest jednym z klasycznych problemów rozwiązywanych za pomocą programowania dynamicznego. Jego złożoność obliczeniowa to O((2^N)*(N^2)).

1.2.3 Maski bitowe

Maska bitowa – jest to wybór pojedynczych bitów lub pól z kilku bitów ciągu danej liczby. Operacje te działają na ciągach bitowych w ten sposób, że bity na odpowiadających sobie miejscach są poddawane operacjom logicznym takim jak: AND, OR lub XOR.

1.2.4 Rozpatrywanie działania algorytmu dla konkretnych danych

Macierz 3x3

|-1 49 79|

|60 -1 91|

|87 8 -1|

Tablica podproblemów:

|-1 -1 -1 -1 -1 -1 -1 -1| //podproblemy dla miasta 1 (maska 6 → 110)

|60 -1 -1 -1 -1 -1 -1 -1| //podproblemy dla miasta 2 (maska 4 → 100)

|87 -1 -1 -1 -1 -1 -1 -1| //podproblemy dla miasta 3 (maska 2 → 010)

Ustawiam na: 6 Maska: 6 subUstawienie: 6 Miasto= 1--> omijam, bo nie może iść do siebie samego (z miasta 1 → 1)

Ustawiam na: 6 Maska: 5 subUstawienie: 4 Miasto= 2 // (miasto 1 → 2)

Wartość do policzenia = miasto: od [1] do [2] = 49 + reszta drogi (1,4) //wchodzę do miasta 2

Ustawiam na: 4 Maska: 6 subUstawienie: 4 Miasto= 1--> omijam // (miasto 2 → 1)

Ustawiam na: 4 Maska: 5 subUstawienie: 4 Miasto= 2--> omijam // (miasto 2→2)

Ustawiam na: 4 Maska: 3 subUstawienie: 0 Miasto= 3 // (miasto 2→3) wchodzę

Wartość do policzenia = miasto: od [2] do [3] =91 + reszta drogi (2,0)

Wartość policzona = miasto: od [2] do [3] = 91 + reszta drogi (2,0) = 87 = 178

Podproblem [1][4] = 178 // Zapisuje pełną drogę do tablicy podproblemów w celu późniejszego porównania z innymi drogami

Tablica podproblemów:

|-1 -1 -1 -1 -1 -1 -1 -1 |

|60 -1 -1 -1 178 -1 -1 -1|

|87 -1 -1 -1 -1 -1 -1 -1 |

Wartość policzona = miasto: od [1] do [2] = 49 + reszta drogi (1,4) = 178 = 227

// Policzona Droga 1→2→3→1

Ustawiam na: 6 Maska: 3 subUstawienie: 2 Miasto= 3 // (miasto 1→3)

Wartość do policzenia: miasto: od [1] do [3] =79 + reszta drogi (2,2)

Ustawiam na: 2 Maska: 6 subUstawienie: 2 Miasto= 1--> omijam // (miasto 3→1)

Ustawiam na: 2 Maska: 5 subUstawienie: 0 Miasto= 2 // (miasto 3→2)

Wartość do policzenia = miasto: od [3] do [2] =8 + reszta drogi (1,0)

Wartość policzona = miasto: od [3] do [2] = 8 + reszta drogi (1,0) = 60 = 68

Ustawiam na: 2 Maska: 3 subUstawienie: 2 Miasto= 3--> omijam

Podproblem [2][2] = 68

Tablica podproblemów:

|-1 -1 -1 -1 -1 -1 -1 -1 |

|60 -1 -1 -1 178 -1 -1 -1 |

|87 -1 68 -1 -1 -1 -1 -1 |

Wartość policzona: miasto: od [1] do [3] = 79 + reszta drogi (2,2) = 68 = 147

// Policzona Droga 1→3→2→1

Podproblem [0][6] = 147

Tablica podproblemów:

|-1 -1 -1 -1 -1 -1 147 -1 |

|60 -1 -1 -1 178 -1 -1 -1 |

|87 -1 68 -1 -1 -1 -1 -1 |

// Drogi całkowite zostają porównane i wybrana zostaje najkrótsza droga

Wynik = 147

Najkrótsza możliwa droga: 1-->3-->2-->1

W taki sam sposób jest zapamiętywana droga między miastami do wypisywana na ekranie (używana jest tablica to nawracania (ang. Backtracking).

1.2.4 Opis implementacji algorytmu

Do zaimplementowania algorytmu wykorzystałem tablice dwuwymiarową wczytywaną z pliku. Podproblemy i nawracanie drogi również zostało zaimplementowane z użyciem tablica dwuwymiarowych generowanych w zależności od wielkości problemu (Tablica[rozmiar][2^rozmiar]). W celu poruszania się między miastami i sprawdzania kolejnych ścieżek zostały użyte operacje AND na maskach bitowych. Droga przebyta przez komiwojażera jest przechowywana w liscie. Całość jest obsługiwana przez aplikację okienkową w języku Java.

1.3 Wstęp teoretyczny dla metody Brute Force

Algorytm przeglądu zupełnego (ang. brute force) polega na sprawdzeniu wszystkich możliwych przypadków, oraz wybraniu tego o najlepszej wartości. Zaletą tego algorytmu jest to, że otrzymany wynik jest najlepszym rozwiązaniem problemu. Wadą jest jednak złożoność czasową wynoszącą O(n!), co w praktyce czyni ten algorytm bezużytecznym dla większych zbiorów danych (w przypadku problemu komiwojażera >12 miast).

2 Plan eksperymentu

2.1 Rozmiar używanych struktur danych

Dla programowania dynamicznego użyte zostaną macierze(miasta) o wielkościach: 2,3,4,5,6,7,8,9,10,12,14,16,18

Dla przeglądu zupełnego zostaną użyte macierze(miasta) o wielkościach: 2,3,4,5,6,7,8,9,10

2.2 Sposób generowania danych

Dane będą generowane za pomocą pętli, która ma za zadanie wykonać dany algorytm 100 razy dla podanej wielkości macierzy z losowymi wartościami, a następnie obliczyć średni czas ich wykonywania.

2.3 Metoda pomiaru czasu

Czas będzie mierzony za pomocą funkcji: System.nanoTime(). Czas będzie mierzony w nanosekundach. Końcowy wynik to uśredniony czas ze 100 instancji podanego problemu.

2.4 Sposób przedstawiania wyników

Wyniki będą przedstawiane za pomocą wykresów. Rozpatrywane będą:

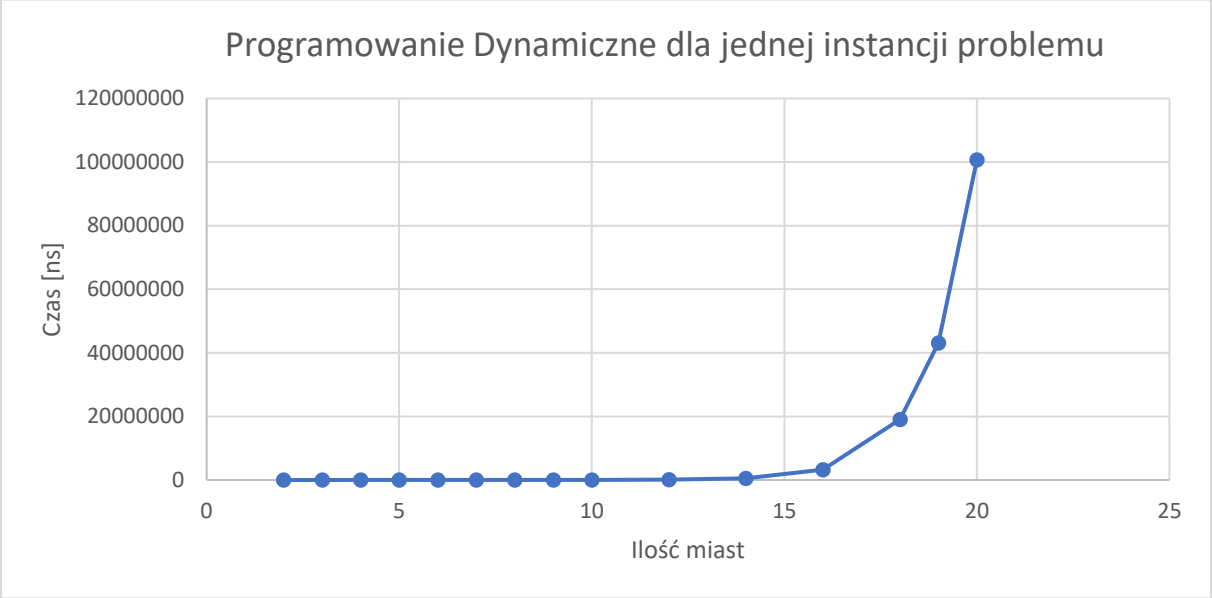
- wykres czasowy dla problemu komiwojażera z użyciem programowania dynamicznego w zależności od ilości miast.

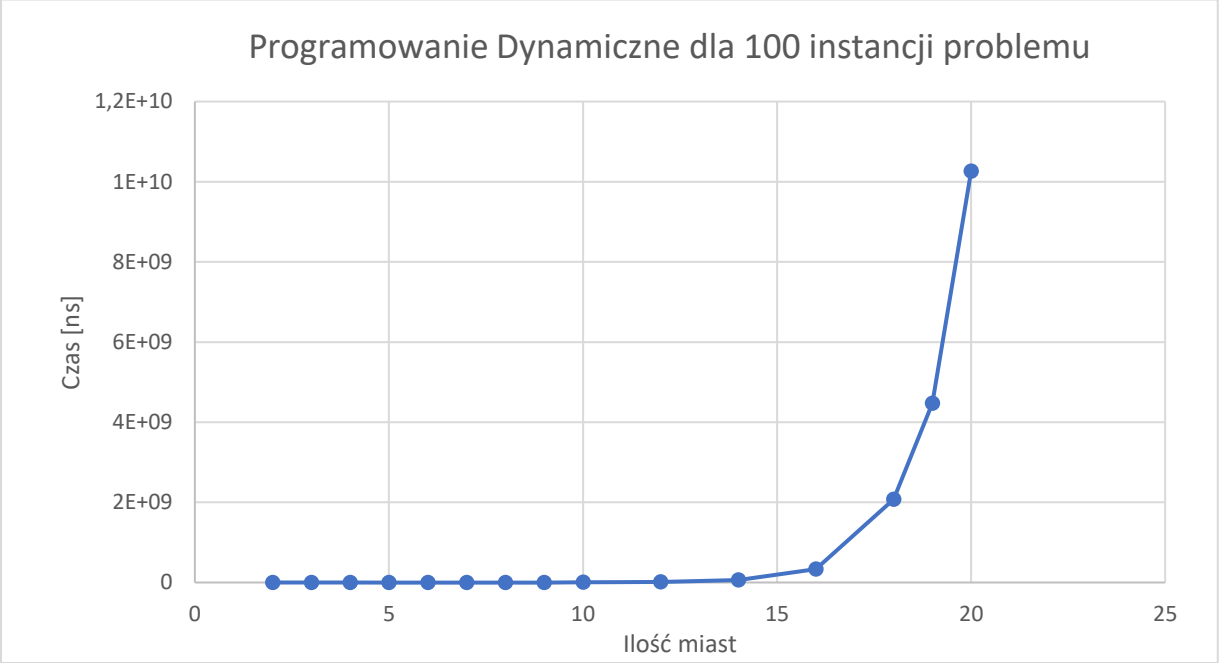
- wykres czasowy dla problemu komiwojażera z użyciem metody przeglądu zupełnego w zależności od ilości miast.

- wykres czasowy porównujący obie metody w zależności od ilości miast.

2.5 Wyniki eksperymentu i wnioski

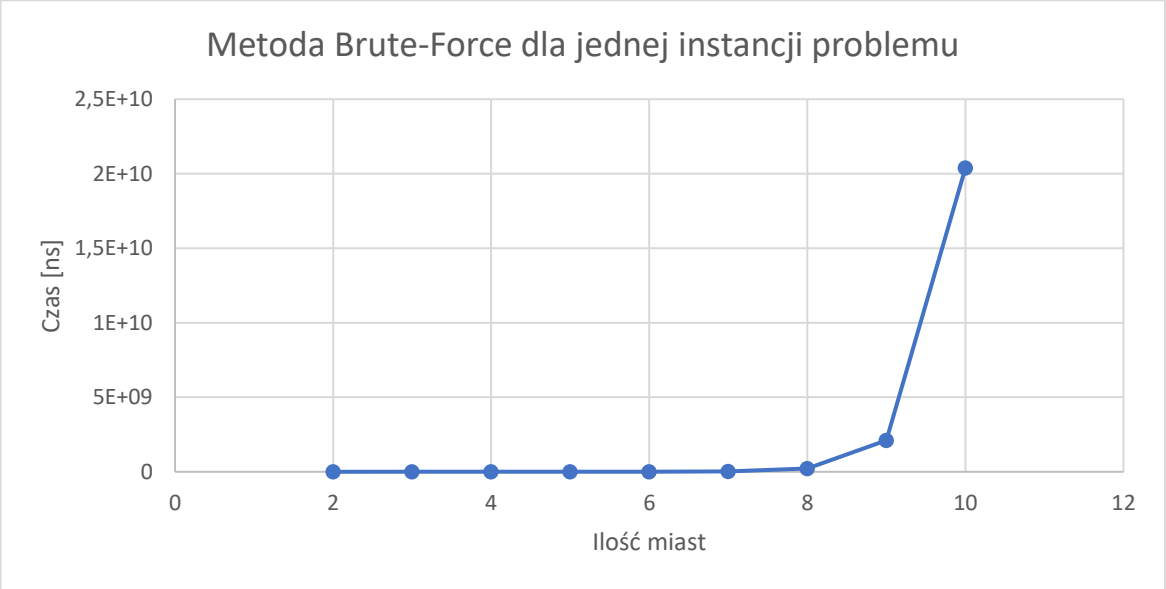
Programowanie Dynamiczne - 1 Instancja problemu		Programowanie Dynamiczne - 100 Instancji problemu	
Ilość miast	Czas Wykonania Algorytmu (ns)	Ilość miast	Czas Wykonania Algorytmu (ns)
2	11	2	232
3	23	3	572
4	59	4	2550
5	303	5	10232
6	568	6	29985
7	2151	7	87714
8	2921	8	249292
9	7480	9	659618
10	16690	10	1673199
12	100895	12	10239179
14	577690	14	59178870
16	3237899	16	330274900
18	19040050	18	2073690780
19	43142017	19	4467647416
20	100689679	20	10258349909

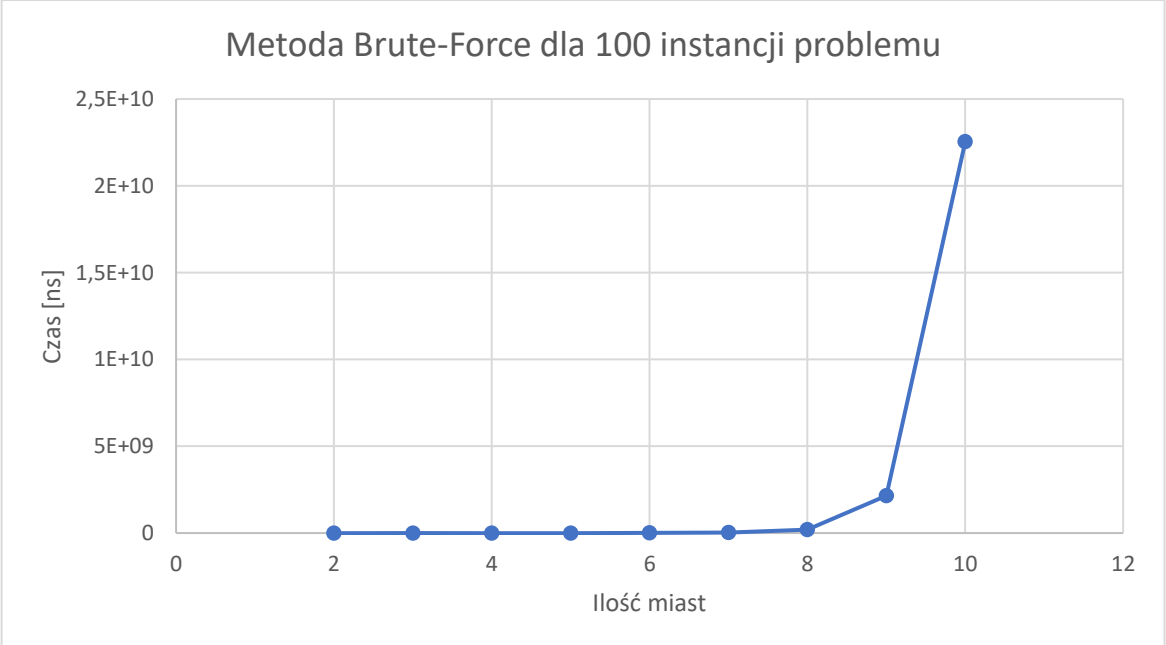




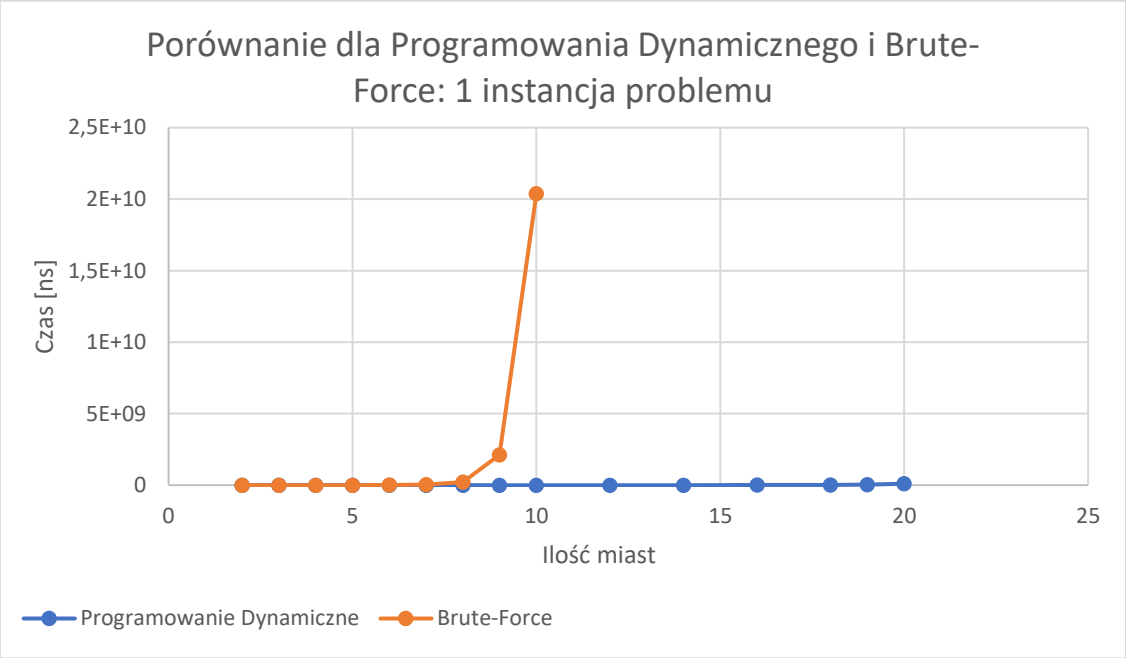
Wnioskując po danych i wykresach stworzonych na ich podstawie można dojść do wniosku, że złożoność obliczeniowa Problemu Komiwojażera dla programowania dynamicznego zgadza się z uzyskanymi przez użytkownika czasami. Dla ilości miast < 10 czas znalezienia najkrótszej drogi jest stosunkowo, krótki. Dla ilości > 10 czas zaczyna drastycznie rosnąć jednak do 20 miast można znaleźć drogę w stosunkowo krótkim czasie (maksymalnie kilka minut). Obliczenie czasu dla jednej instancji pokrywa się ze średnim czasem dla 100 instancji tego samego problemu.

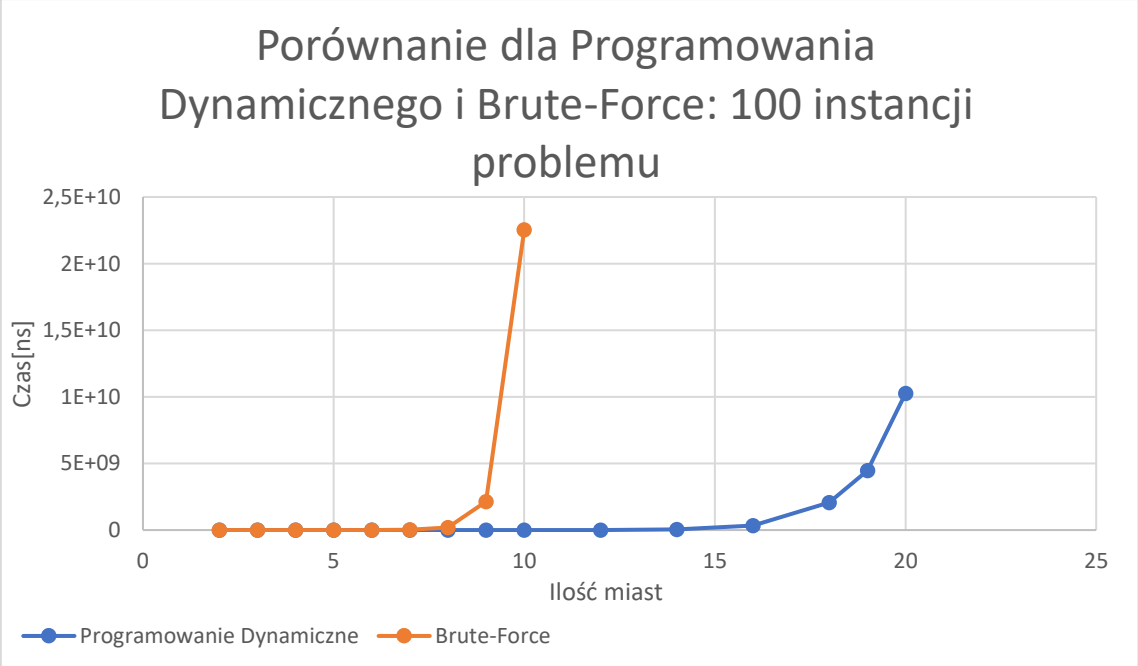
Metoda Brute-Force - 1 instancja problemu		Metoda Brute-Force - 100 instancji problemu	
Ilość miast	Czas Wykonania Algorytmu (ns)	Ilość miast	Czas Wykonania Algorytmu (ns)
2	13247	2	105779
3	40091	3	35745
4	176570	4	162245
5	709949	5	584610
6	3518938	6	3292581
7	32039454	7	25459613
8	214478520	8	198125644
9	2116486805	9	2142814325
10	20393809621	10	22542548721





Wnioskując po danych i wykresach dla metody przeglądu zupełnego można dojść do podobnych konkluzji jak w przypadku programowania dynamicznego z tą różnicą, że czas znalezienia najkrótszej drogi zaczyna rosnąć dużo szybciej przy ilości miast > 8 co czyni tę metodę dużo wolniejszą. Algorytm Brute-Force dla ilości miast > 12 zaczyna obliczać drogę bardzo długo (około kilkudziesięciu minut). Wykres dla jednej instancji pokrywa się z dużą dokładnością względem wykresu dla 100 instancji tego samego problemu.





Obserwując porównanie wyników czasowych dla obu metod znajdowania drogi w problemie komiwojażera można dojść do wniosku, że programowanie dynamiczne pozwala na o wiele szybsze znalezienie drogi niż używając metody przeglądu zupełnego. Jednocześnie umożliwia przetestowanie algorytmu dla dwukrotnie większej ilości miast w rozsądnym przedziale czasowym (do kilku minut).

3 Wnioski końcowe

Algorytmy programowania dynamicznego i przeglądu zupełnego zostały zaimplementowane poprawnie, a wykresy czasowe potwierdzają złożoność czasową dla problemu komiwojażera. Ze względu na użycie języka Java pierwsze uruchomienia algorytmów dla różnych danych dawały zakłamane wyniki, przez co za każdym razem po uruchomieniu programu należało kilka razy „testowo" uruchomić algorytm dla małej instancji np. 2 miast, a dopiero następnie możliwe było uzyskanie zadowalających wyników. Powodem takiej sytuacji jest natura języka Java, który używa maszyny wirtualnej.