Projekt z Projektowania Efektywnych Algorytmów

Numer etapu: 1

Zadanie wykonał: Michał Wróbel (259132)

Grupa laboratoryjna: K00-490

Termin zajęć: WT 17..05

Data oddania: 22.11.2022

Prowadzący: Dr inż. Jarosław Mierzwa

1. **Cel:**

   Celem projektu było zaimplementowanie algorytmów rozwiązujących asymetryczny problem komiwojażera (wybrane przeze mnie metody to przegląd zupełny oraz programowanie dynamiczne (algorytm Helda-Karpa)). Algorytmy zostaną zaimplementowane w języku Python.

2. **Wstęp teoretyczny:**

   Zaimplementowane algorytmy miały za zadanie odnalezienie minimalnego cyklu Hamiltona, czyli drogi rozpoczynającej się z zadanym wierzchołku grafu, która przechodzi tylko raz przez wszystkie pozostałe wierzchołki oraz wraca do wierzchołka początkowego Graf jest przechowywany w postaci macierzy kwadratowej, w której poszczególne wartości są wagami drogi pomiędzy poszczególnymi miastami.

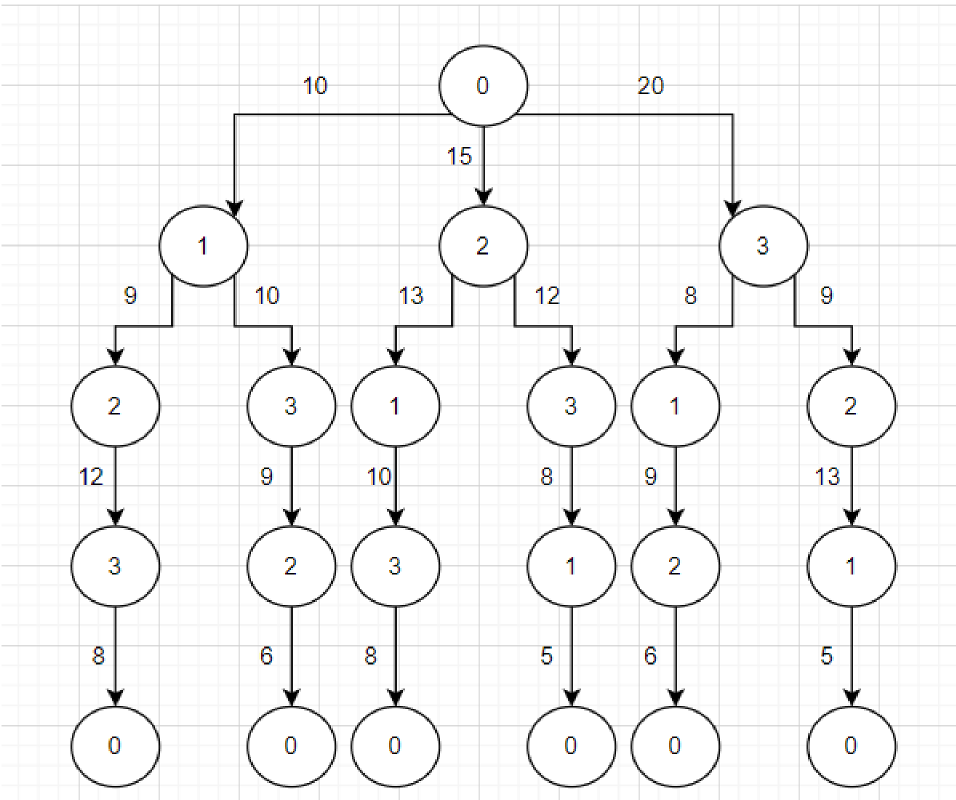
   W metodzie przeglądu zupełnego musimy rozpatrzyć wszystkie kombinacje tworzące cykl Hamiltona w grafie. Liczba tych kombinacji to (n-1)! Rozpatrywanie każdego takiego przypadku prowadzi do wykładniczej złożoności obliczeniowej: O(n!).

   Algorytm Helda-Karpa jest oparty na programowaniu dynamicznym, czyli podziale jednego problemu na pod problemy ze względu na określone parametry. Jego złożoność obliczeniowa to O(n^2 2^n).

3. **Przykład praktyczny dla problemu Helda-Karpa.**

   Dany jest graf zawierający następujące wierzchołki:

	0	1	2	3
0	0	10	15	20
1	5	0	9	10
2	6	13	0	12
3	8	8	0	0



$$g(i,s) = \min k \ \in s\, \{c_{ik} + g(k, s - \{k\})\}$$

1 poziom:
g(1, 0) = 5
g(2, 0) = 6
g(3, 0) = 8

2 poziom:
g(1, {2, 0}) = 15
g(1, {3, 0}) = 18
g(2, {1, 0}) = 18
g(2, {3, 0}) = 20
g(3, {1, 0}) = 13

g(3, {2, 0}) = 15

3 poziom:
g(1, {3, 2, 0}) = 25 <= min
g(1, {2, 3, 0}) = 29

g(2, {1, 3, 0}) = 31
g(2, {3, 1, 0}) = 25 <= min

g(3, {1, 2, 0}) = 23 <= min
g(3, {3, 1, 0}) = 27

4 poziom:
g(0, {1, 3, 2, 0}) = 35 <= min
g(0, {2, 3, 1, 0}) = 40
g(0, {3, 1, 2, 0}) = 43

4. Opis implementacji algorytmu

Przeglądu zupełnego:

```python
def brute_force_algorithm(matrix: int):
    #Miejsce w którym będziemy przechowywać wynik
    result = (float('inf'),)

    #Dla każdej permutacji zbioru miast wykonujemy następujące operacje
    for permutation in permutations(range(len(matrix))):
        permutation += (permutation[0],)

        #Obliczamy całkowity koszt podróży dla danej permutacji zbioru
        cost = 0
        for i in range(len(permutation) - 1):
            cost += matrix[permutation[i]][permutation[i + 1]]

        #Jeżeli koszt jest niższy od poprzedniego to zapisujemy wynik (koszt oraz ścieżkę) w result
        if cost < result[0]:
            result = (cost, permutation)

    #Po zakończeniu pętli zwracamy result
    return result
```

Helda-Karpa_v1:

```python
def algorithm(current_city: int, unvisited_cities: int, matrix: int):

    # Jeżeli wszystkie miasta zostały odwiedzone rozpoczynamy zwijanie rekurencji
    if len(unvisited_cities) == 0:
        # Zwracamy koszt podróży od naszego aktualnego miasta do miasta początkowego
        return matrix[current_city][0]

    all_possible_paths = []
    # Dla każdego nie odwiedzonego miasta wykonujemy następujące operacje
    for next_city in unvisited_cities:
        # Do listy wszystkich możliwych ścieżek dodajemy koszt podróży z miasta w którym aktualnie jesteśmy oraz wywołujemy ponownie
        # naszą funkcję, która zwróci nam koszt podróży do kolejnych miast.
        cost = matrix[current_city][next_city] + algorithm(next_city, unvisited_cities - {next_city}, matrix)
        all_possible_paths.append(cost)

    # Zwracamy minimalną wartość z listy wszystkich możliwych ścieżek z danego miasta
    result = min(all_possible_paths)

    return result


def held_karp_algorithm(matrix: int):

    # Obliczamy ilość miast oraz wszystke nie odwiedzone jeszcze miasta
    matrix_length = len(matrix)
    unvisited_cities = set(range(1, matrix_length))

    # Pierwsze wywołanie funkcji rekurancyjnej (jako argumenty podajemy miasto startowe, wszystkie nie odwiedzone miasta macierz)
    result = algorithm(0, unvisited_cities, matrix)

    return result
```

Helda-Karpa_v2:

```python
def algorithm_v2(current_city: int, unvisited_cities: int, matrix: int):

    # Jeżeli wszystkie miasta zostały odwiedzone rozpoczynamy zwijanie rekurencji
    if len(unvisited_cities) == 0:
        # Zwracamy koszt podróży od naszego aktualnego miasta do miasta początkowego
        path = [0, current_city]
        result = [matrix[current_city][0], path]
        return result

    all_possible_paths = []
    # Dla każdego nie odwiedzonego miasta wykonujemy następujące operacje
    for next_city in unvisited_cities:
        # Do listy wszystkich możliwych ścieżek dodajemy koszt podróży z miasta w którym aktualnie jesteśmy oraz wywołujemy ponownie
        # naszą funkcję, która zwróci nam koszt podróży do kolejnych miast.
        current_cost = matrix[current_city][next_city]
        next_cost = algorithm_v2(next_city, unvisited_cities - {next_city}, matrix)
        cost = current_cost + next_cost[0]
        path = next_cost[1] + [current_city]
        all_possible_paths.append([cost, path])

    # Konwertujemy tablicę na tablicę numpy w celu wyszukania najkrótszej ścieżki.
    all_possible_paths_numpy = numpy.array(all_possible_paths, dtype=object)

    # Zwracamy minimalną wartość podróży oraz ścieżkę z listy wszystkich możliwych ścieżek z danego miasta
    result = all_possible_paths_numpy[numpy.argmin(all_possible_paths_numpy[:, 0]), :]

    return result
```

```python
def held_karp_algorithm_v2(matrix: int):

    # Obliczamy ilość miast oraz wszystke nie odwiedzone jeszcze miasta
    matrix_length = len(matrix)
    unvisited_cities = set(range(1, matrix_length))

    # Pierwsze wywołanie funkcji rekurancyjnej (jako argumenty podajemy miasto startowe, wszystkie nie odwiedzone miasta macierz)
    result = algorithm_v2(0, unvisited_cities, matrix)

    # Odwracamy kolejność ścieżki
    result[1].reverse()

    return result
```

Wersja v2 dodatkowo wyświetla dodatkowo ścieżkę Hamiltona, oprócz samego wyniku. Prawdopodobnie ze względu na złą optymalizację (prawdopodbnie można było lepiej zaimplementować tablicę przechowującą ścieżkę), algorytm ten jest zdecydowanie wolniejszy od swojej wersji, która wraca tylko sam koszt podróży. Do testów została użyta wersja v1 algorytmu (bez wracania ścieżki). Oba te algorytmy działają na tej samej zasadzie – rekurencyjnie rozwijają drzewo rozwiązań a następnie je zwijają wybierając minimalne wartości podróży.

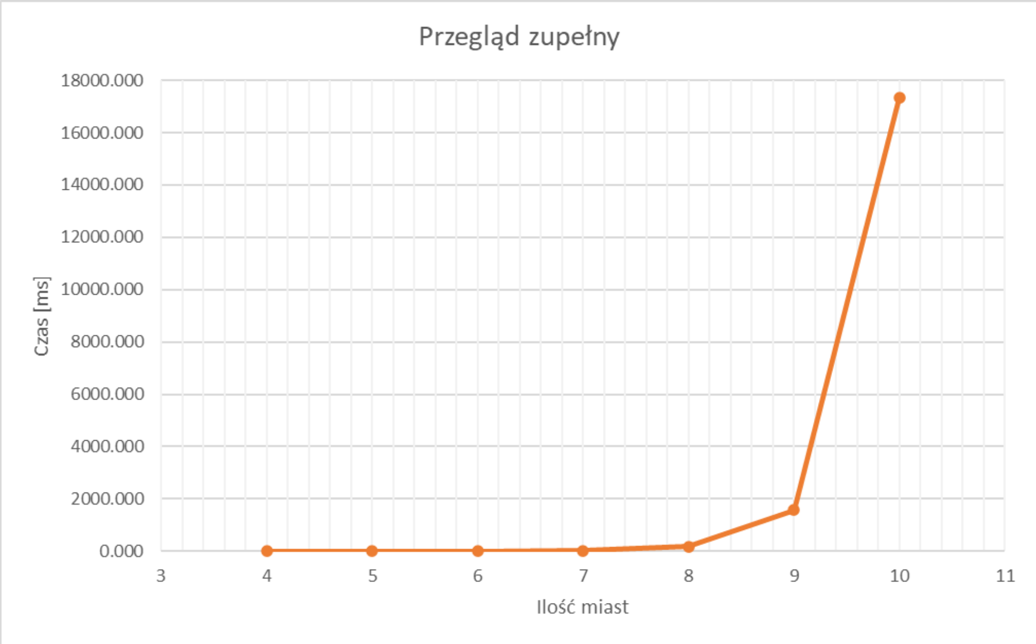
5. Plan eksperymentu

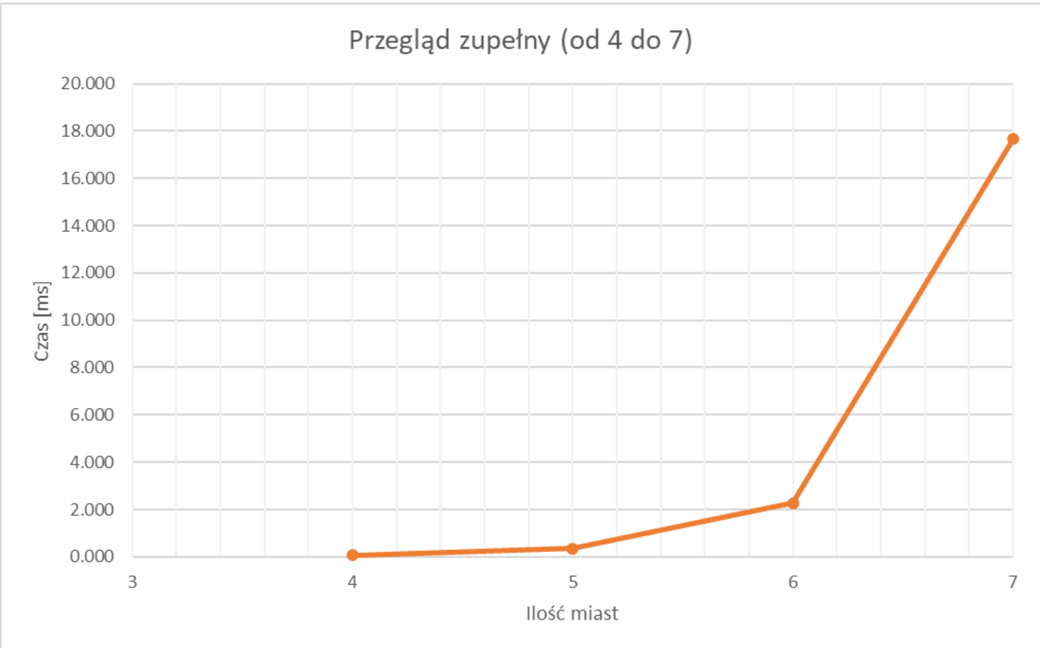
W celu przetestowania efektywności algorytmów dla każdego z nich zostanie wygenerowanych 150 losowych instancji dla następujących liczb miast: 4, 5, 6, 7, 8, 9, 10. Pomiary zaczynamy wykonywać zgodnie z zaleceniami dopiero od 50 instancji, ponieważ językiem użytym do zaimplementowania algorytmów był Python. Pomiary czasu zostaną wykonane przy pomocy modułu time. Dla algorytmu Held-Karpa została użyta do testów pierwsza wersja algorytmu (zwracająca sam wynik, bez ścieżki).

6. Wynik eksperymentu

Przegląd zupełny

Przegląd zupełny		
Ilość miast	czas [ns]	czas [ms]
4	61899.000	0.062
5	343978.000	0.344
6	2275623.000	2.276
7	17677018.000	17.677
8	171752120.000	171.752
9	1582126540.000	1582.127
10	17331260760.000	17331.261

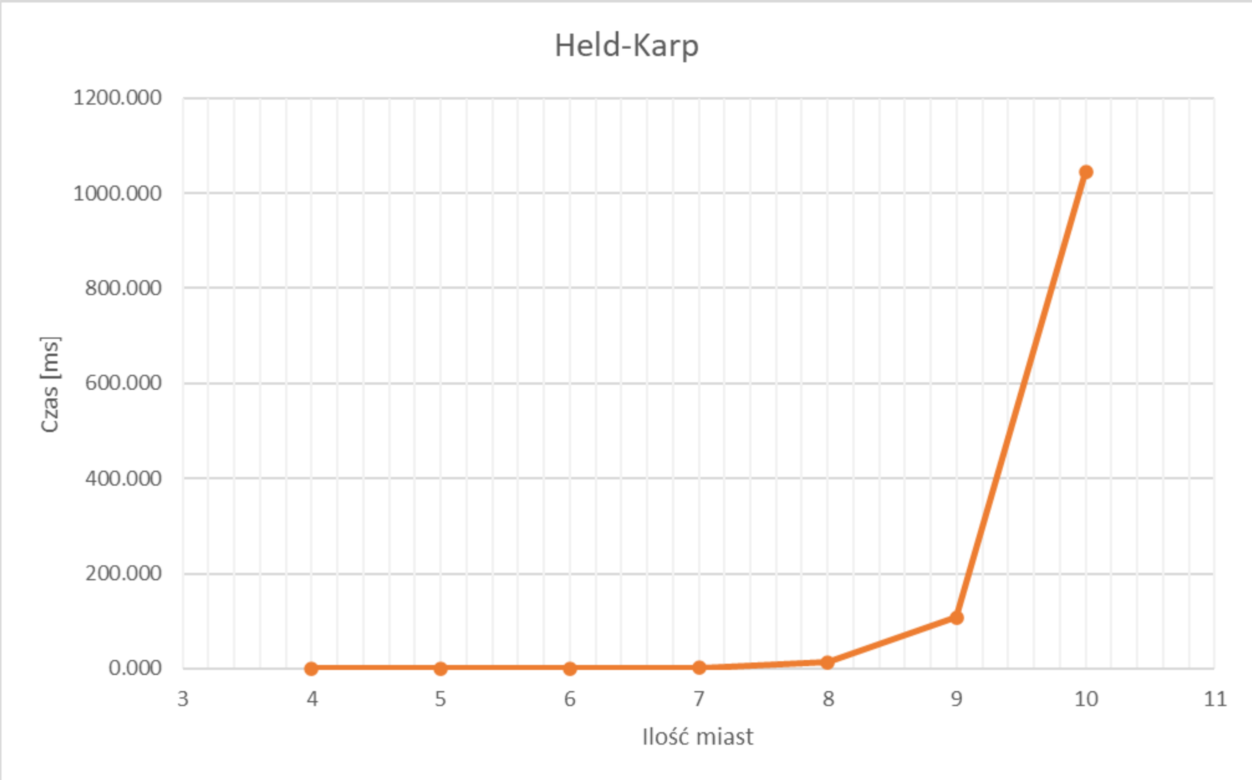


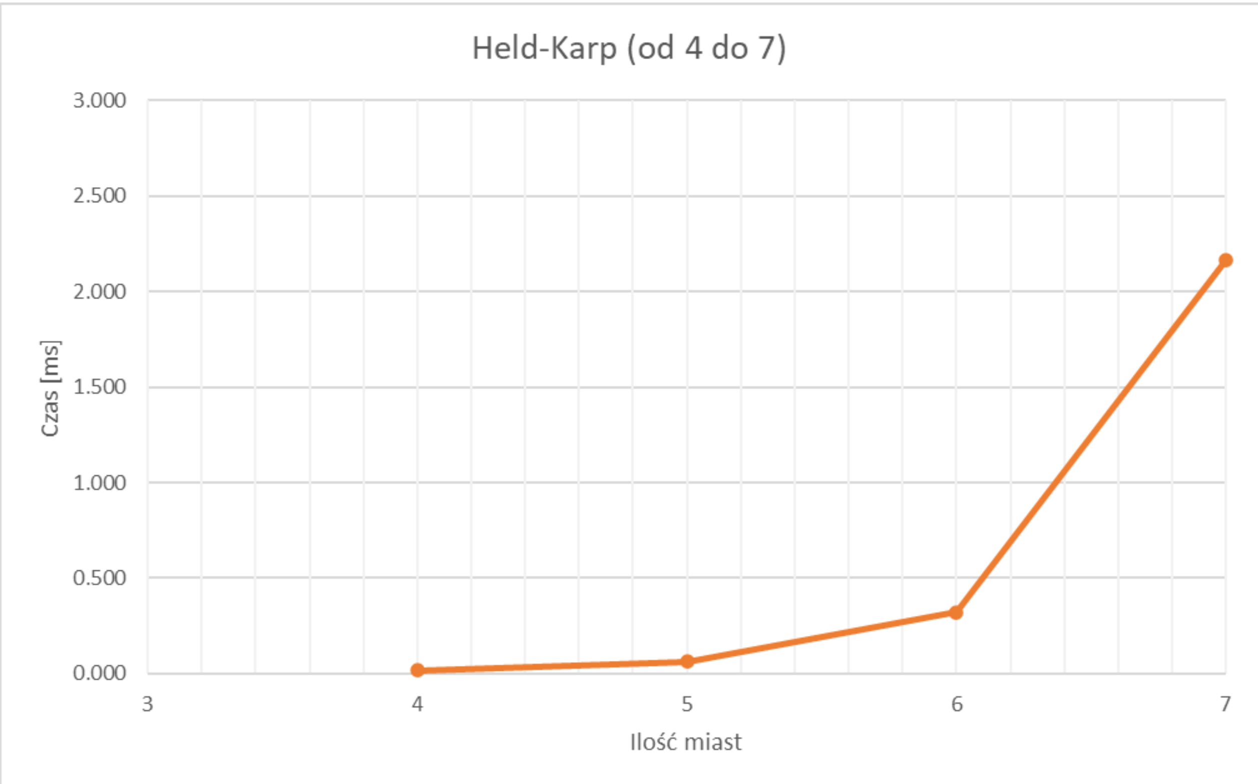


Algorytm Helda-Karpa

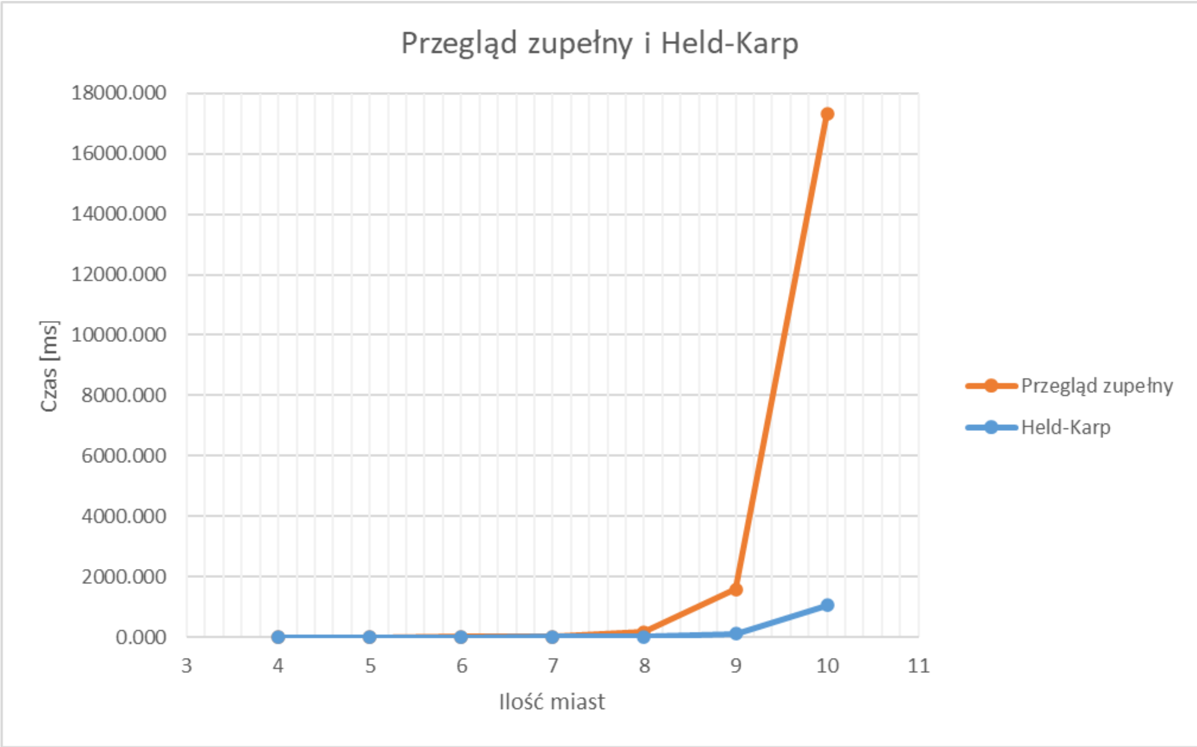
Held-Karp		
Ilość miast	czas [ns]	czas [ms]
4	17486.000	0.017
5	62716.000	0.063
6	321559.000	0.322
7	2165495.000	2.165
8	13732309.000	13.732

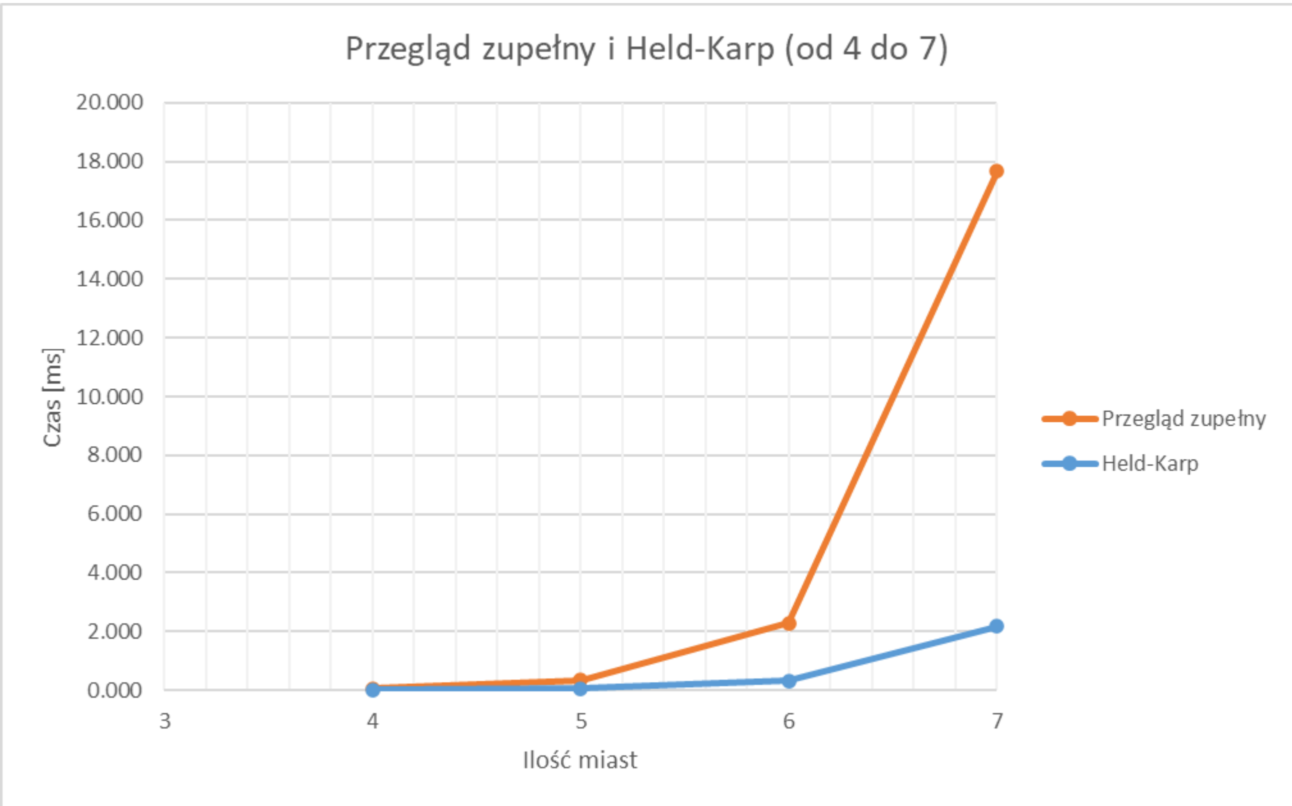
9	107988385.000	107.988
10	1045115076.000	1045.115





Porównanie przeglądu zupełnego oraz Held-Karpa


Przegląd zupełny i Held-Karp
18000.000
16000.000
14000.000
12000.000
10000.000
8000.000
6000.000
4000.000
2000.000
0.000
Czas [ms]
3
4
5
6
7
8
9
10
11
Ilość miast
Przegląd zupełny
Held-Karp


Przegląd zupełny i Held-Karp (od 4 do 7)
20.000
18.000
16.000
14.000
12.000
10.000
8.000
6.000
4.000
2.000
0.000
Czas [ms]
3
4
5
6
7
Ilość miast
Przegląd zupełny
Held-Karp

7. Wnioski:

Wykres dla metody przeglądu zupełnego wydaje się potwierdzać jego teoretyczną złożoność obliczeniową - O(n!). Z tym samym mamy do czynienia w przypadku algorytmu Helda-Karpa opartego na programowaniu dynamicznym (jego teoretyczna złożonośc obliczeniowa to O(n^2 2^n)).

Zgodnie z przewidywaniami algorytm przeglądu zupełnego jest dużo mniej wydajny od Helda-Karpa (szczególnie dużą różnicę widać od macierzy dla 7 miast wzwyż), nawet przy stosunkowo małych macierzach (np. dla 4 miast).