Sprawozdanie z projektu z kursu Projektowanie Efektywnych Algorytmów

Autor: Piotr Kołeczek

Numer albumu: 234940

Prowadzący projekt: Dr inż. Dariusz Banasiak

Grupa zajęciowa: środa 18:55 – 20:35

Termin oddania: 28.10.2019r.

Temat projektu: Implementacja i analiza algorytmów dokładnych rozwiązujących asymetryczny problem komiwojażera (ATSP).

1. Wprowadzenie do problemu oraz wstęp teoretyczny.

Rozważanym problemem projektowym jest programowa implementacja asymetrycznego problemu komiwojażera. Ów problem polega na wyznaczeniu takiego cyklu Hamiltona w grafie pełnym ważonym, którego koszt jest możliwie jak najmniejszy. Problem ilustrowany jest często jako problem podróżnika, który ma za zadanie przejść przez wszystkie miasta w krainie przebywając możliwie jak najkrótszą drogę (lub poświęcając na podróż jak najmniej czasu).

Problem projektowy możemy rozważyć w przypadku, gdy droga z miasta **A** do miasta **B** jest taka sama jak z miasta **B** do miasta **A** oraz kiedy w przeciwnym kierunku **może** mieć inną wartość. Ta różnica dzieli problem na symetryczny (STSP) oraz asymetryczny problem komiwojażera (ATSP).

Problem komiwojażera jest problemem, który zalicza się do tak zwanych problemów *NP-trudnych,* czyli takich, dla których nie wynaleziono algorytmów, które rozwiązują ów problem w sposób optymalny o wielomianowej złożoności obliczeniowej. Algorytmy, które wyszukują optymalny cykl Hamiltona w grafie dla większych instancji grafów są bardzo złożone obliczeniowo i są bardzo czasochłonne, dlatego też często wykorzystuje się algorytmy liczące przybliżoną wartość optymalnej drogi.

2. Metody rozwiązania.

Przyjrzyjmy się najpierw najpopularniejszemu algorytmowi dokładnemu, który rozwiązuje problem projektowy. Jest to algorytm **Brute Force**, który po polsku nazywamy algorytmem *przeglądu zupełnego*, *siłowym, brutalnej siły.* Najtrafniejsze stwierdzenie to algorytm przeglądu zupełnego, ponieważ metoda Brute Force polega na odnalezieniu wszystkich cykli Hamiltona w rozważanym grafie, a następnie wybranie tego, którego koszt cyklu jest optymalny.

Niewątpliwą zaletą tego algorytmu jest fakt, że jest on dokładny, a zatem gdy zakończy się wykonywanie algorytmu Brute Force możemy być pewni, że znalazł on optymalne rozwiązanie problemu.

Pokrótce mówiąc działa on w taki sposób, że przeglądane są kolejne wierzchołki grafu do momentu dotarcia do końca (wierzchołka startowego), co zamyka cykl. W tym momencie sprawdza czy znalezione rozwiązanie jest lepsze od poprzedniego wyszukanego. Jeżeli tak, aktualny najlepszy cykl jest nadpisywany nowym (lepszym) i algorytm wraca ponownie do miejsca startu, wykonując czynność od nowa. Czynność ta jest powtarzana do momentu, kiedy wszystkie możliwe drogi w grafie zostaną przejrzane. I w tym momencie, niejawnie zresztą, odkryliśmy największą wadę tego algorytmu. Działanie algorytmu w taki sposób powoduje wytworzenie niesłychanie wielkiej liczby możliwych rozwiązań do przejrzenia. To rzutuje bezpośrednio na czas wykonywania algorytmu, którego złożoność wynosi:

$$O(n!)$$

co oznacza, że algorytm Brute Force jest jednym z najwolniejszych algorytmów, gdyż mamy do czynienia ze złożonością wykładniczą (superwykładniczą).

Inną metodą, którą rozważymy jest metoda programowania dynamicznego. Jest to taka strategia działania, w której jeden duży problem optymalizacyjny dzielimy na mniejsze pod-problemy, łatwiejsze obliczeniowo. Określamy stan, który będzie podzbiorem zbioru wszystkich wierzchołków z wyszczególnionym początkiem i końcem.

Taka reprezentacja dużego problemu pozwala nam na optymalizację tego podejścia – skoro ścieżka naszego kupca ma być cyklem i ma przechodzić przez wszystkie wierzchołki, to bez straty ogólności możemy założyć, że zawsze zaczyna się ona w pewnym, wyszczególnionym wierzchołku (nazwijmy go wierzchołkiem początkowym). W ten sposób przestrzeń stanów można ograniczyć do podzbiorów zbioru wierzchołków, wraz z wyszczególnionym wierzchołkiem końcowym.

Dzięki programowaniu dynamicznemu złożoność czasową możemy zredukować do postaci:

$$O(n^2 \cdot 2^n)$$

A taka złożoność w porównaniu do metody przeglądu zupełnego, choć dalej ponad-wykładnicza, jest znacznie mniej zajmująca czasowo.

3. Metodyka pomiarowa oraz pomiary czasowe.

Każdy z wyżej wymienionych algorytmów został poddany seryjnym pomiarom czasowym. Ze względu na fakt, iż algorytmy dokładne rozwiązujące problem komiwojażera mają ponad-wykładniczą złożoność czasową oraz obliczeniową, niemożliwym było testowanie algorytmów dla większych instancji grafów. Dlatego też do testów zostały wykorzystane małe instancje problemów przygotowane przez doktora Mierzwę oraz magistra Idzikowskiego (pliki z grafami zostały pobrane ze stron wspomnianych prowadzących). Górną granicę czasową testowania poprawności algorytmów przyjąłem na **1:30** minut. Wszelkie zmienne oraz struktury danych wykorzystywały 32-bitowe liczby całkowite. Dla algorytmu przeglądu zupełnego (Brute Force) zostało wykonanych **1000** prób pomiarowych w przypadku macierzy o rozmiarze N = 3, 4, 6, po **200** iteracji dla macierzy N = 10 oraz po **2** iteracje dla N = 12 oraz 13.

Dla programowania dynamicznego ze względu na bardzo szybkie operacje logiczne na liczbach oraz rozdzielenie problemu na mniejsze pod-problemy testowanie algorytmu rozpocząłem dla instancji grafów o rozmiarze od **14** wierzchołków. Dla instancji o rozmiarze N = 14, 15, 16 wykonano **1000** testów, dla N = 17 – **200** prób pomiarowych, dla N = 18 - **100** prób, N = **20** wykonano **10** iteracji, natomiast dla N = 21, 22, 23, 24 zrobiono kolejno **5, 3, 2** oraz **1** iterację pomiarową czasu.

Dla wszystkich testów wyniki zostały uśrednione.

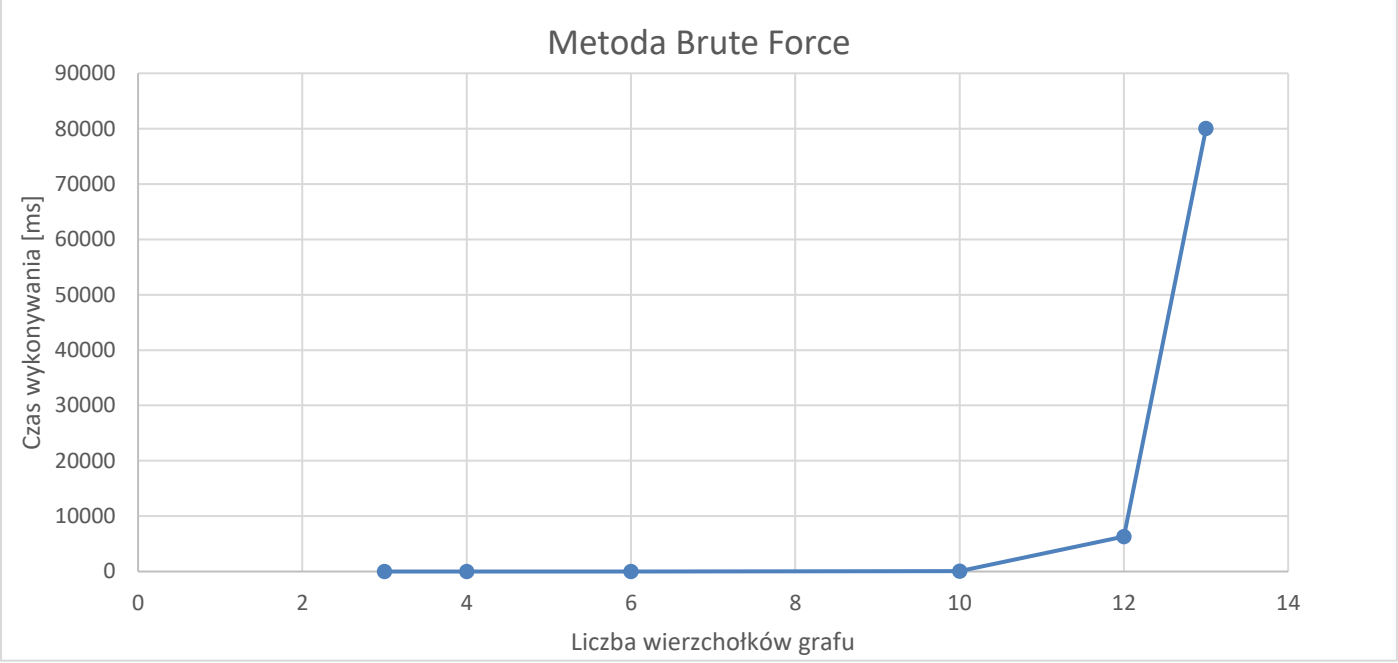
Zadanie projektowe zostało zrealizowane w języku programowania C# w platformie .NET, a do samych pomiarów posłużyłem się precyzyjnym czasomierzem **Stopwatch**, który zawarty jest w przestrzeni **System.Diagnostics.**

3.1 Metoda Brute Force.

Poniżej zostały ukazane uśrednione wyniki dla czasu trwania metody przeglądu zupełnego problemu komiwojażera w postaci tabelarycznej oraz graficznej:

Tabela 1: Zależności czasowe między rozmiarem grafu, a czasem rozwiązywania TSP przy użyciu algorytmu Brute Force.

Nazwa pliku	Liczba wierzchołków grafu	Optymalny koszt cyklu Hamiltona	Uzyskana waga cyklu	Średni czas wykonywania algorytmu w [ms]
tsp_3.txt	3	147	147	0.0373222
tsp_4.txt	4	107	107	0.0397718
tsp_6_2.txt	6	80	80	0.0525811
tsp_10.txt	10	212	212	51.147784
tsp_12.txt	12	264	264	6314.35295
tsp_3.txt	13	269	269	80019.7584



Rysunek 1: Zależność czasu wykonywania algorytmu przeglądu zupełnego w zależności od rozmiaru macierzy grafu.

3.2. Metoda programowania dynamicznego.

W metodzie programowania dynamicznego zastosowano technikę podziału całego problemu na szereg mniejszych pod-problemów – główną ideą tej metody jest obliczenie optymalnego rozwiązania dla wszystkich pod-problemów długości N (N jest wartością określającą liczbę wierzchołków w grafie) korzystając z rekurencyjnie wyliczonych wartości mniejszych pod-problemów o długości N-1.

W rozważanym problemie są potrzebne dwie rzeczy:

- Zestaw odwiedzonych wierzchołków w rozważanej ścieżce,
- Indeks ostatniego odwiedzonego wierzchołka w ścieżce.

Posiadając zbiór N-elementowy, liczba wszystkich podzbiorów danego zbioru wynosi 2^N. Żeby móc zapamiętać aktualny stan ścieżki, czyli nasz zestaw odwiedzonych wierzchołków możemy się posłużyć tak zwaną maską bitową. Maska bitowa o długości N to liczba binarna o N-bitach, w której to każdy bit może zostać ustawiony na 1 lub 0 (1 – wierzchołek odwiedzony, 0 – wierzchołek nieodwiedzony).

3.2.1. Opis implementacji algorytmu.

Zaimplementowany algorytm wykorzystuje dwie tablice o rozmiarze $[N][2^N]$ – jedna służy do przechowywania pod-problemów, które są generowane dla mniejszych podgrafów całego grafu (dla zbiorów M elementowych ze zbioru N, gdzie N jest równe liczbie miast). W celu oznaczania kolejnych wierzchołków, które zostały odwiedzone wykorzystałem 32 bitową maskę (int) i wykonując operacje logiczne AND, OR i XOR. Najlepsza droga jest przechowywana na stosie o własnej implementacji.

3.2.2. Przykład rozwiązania TSP metodą programowania dynamicznego.

Rozważmy macierz kosztów:

	0	**1**	**2**
0	inf	49	79
1	60	Inf	91
2	87	8	inf

Pogrubione liczby w pierwszym wierszu oraz pierwszej kolumnie oznaczają ich numerację.

Tablice pod-problemów dla każdego z wierzchołka:

|inf inf inf inf inf inf inf inf|

|inf inf inf inf inf inf inf inf|

|inf inf inf inf inf inf inf inf|

Rozpoczynamy sprawdzanie od wierzchołka **0.** Początkowa maska odwiedzin wierzchołków: **001**.

Dla wierzchołka **0:**

Czy maska odwiedzin wynosi **111? NIE** (wynosi **001,** przechodzę dalej)

Czy 0 wierzchołek został odwiedzony? **TAK** (omijam)

Czy 1 wierzchołek został odwiedzony? **NIE** (zaczynam go rozpatrywać)

Nowa maska odwiedzin = **011** (wierzchołek **0** oraz **1** odwiedzony)

Wartość podproblemu = macierzKosztów[0][1] + obliczenie kosztu(1, 011 = 3) -> wchodzę do miasta **1**

Przy przejściu z wierzchołka **0** do **1:**

Czy maska odwiedzin wynosi **111? NIE** (wynosi **011,** przechodzę dalej)

Czy 0 wierzchołek został odwiedzony? **TAK** (omijam)

Czy wierzchołek 1 został odwiedzony? **TAK** (omijam)

Czy wierzchołek 2 został odwiedzony? **NIE** (zaczynam go rozpatrywać)

Nowa Maska odwiedzin = **111** (wierzchołek **0, 1, 2** odwiedzony).

Wartość podproblemu = macierzKosztów[1][2] + obliczenie kosztu(2, 111 = 7) -> wchodzę do miasta **2**

Przy przejściu z wierzchołka **1** do **2:**

Czy maska odwiedzin wynosi **111? TAK,** zwracam macierzKosztów[2][0] = **87**

Powrót do wierzchołka **1:**

Wartość podproblemu = macierzKosztów[1][2] + obliczenie kosztu (2, 111 = 7) **=** 91 + 87 = **178**

Podproblem[1][3] = **178**

Tablice podproblemów:

|inf inf inf inf inf inf inf inf|

|inf inf inf 178 inf inf inf inf|

|inf inf inf inf inf inf inf inf|

Powrót do wierzchołka **0:**

Wartość podproblemu = macierzKosztów[0][1] + obliczenie kosztu(1, 011 = 3) = 49 + 178 = **227**

Przy przejściu z wierzchołka **0** do **2:**

Nowa maska = 101 (**0** i **2** wierzchołek odwiedzony)

Wartość podproblemu = macierzKosztów[0][2] + obliczenie kosztu(2, 101 = 5) -> wchodzę do miasta **2**

Dla wierzchołka **2:**

Czy maska odwiedzin wynosi **111? NIE** (wynosi **101,** przechodzę dalej)

Czy 0 wierzchołek został odwiedzony? **TAK** (omijam)

Czy wierzchołek 1 został odwiedzony? **NIE** (zaczynam go rozpatrywać)

Nowa maska odwiedzin = **111** (wierzchołek **0, 1 i 2** odwiedzony)

Wartość podproblemu = macierzKosztów[2][1] + obliczenie kosztu(1, 111 = 7) -> wchodzę do miasta **1**

Dla wierzchołka **1:**

Czy maska odwiedzin wynosi **111? TAK** zwracam macierzKosztów[1][0] = **60**

Powrót do wierzchołka **2:**

Wartość podproblemu = macierzKosztów[2][1] + obliczenie kosztu(1, 111 = 7) = 8 + 60 = **68**

Podproblem[2][5] = **68**

Tablice podproblemów:

|inf inf inf inf inf inf inf inf|

|inf inf inf 178 inf inf inf inf|

|inf inf inf inf inf 68 inf inf|

Wracam do wierzchołka **0:**

Wartość podproblemu = macierzKosztów[0][2] + obliczenie kosztu(1, 101 = 5) = 79 + 68 = **147**

Podproblem[0][1] = **147**

Tablice podproblemów:

|inf 147 inf inf inf inf inf inf|

|inf inf inf 178 inf inf inf inf|

|inf inf inf inf inf 68 inf inf|

Najlepszy obliczony koszt przy odwiedzeniu wszystkich wierzchołków = **147**

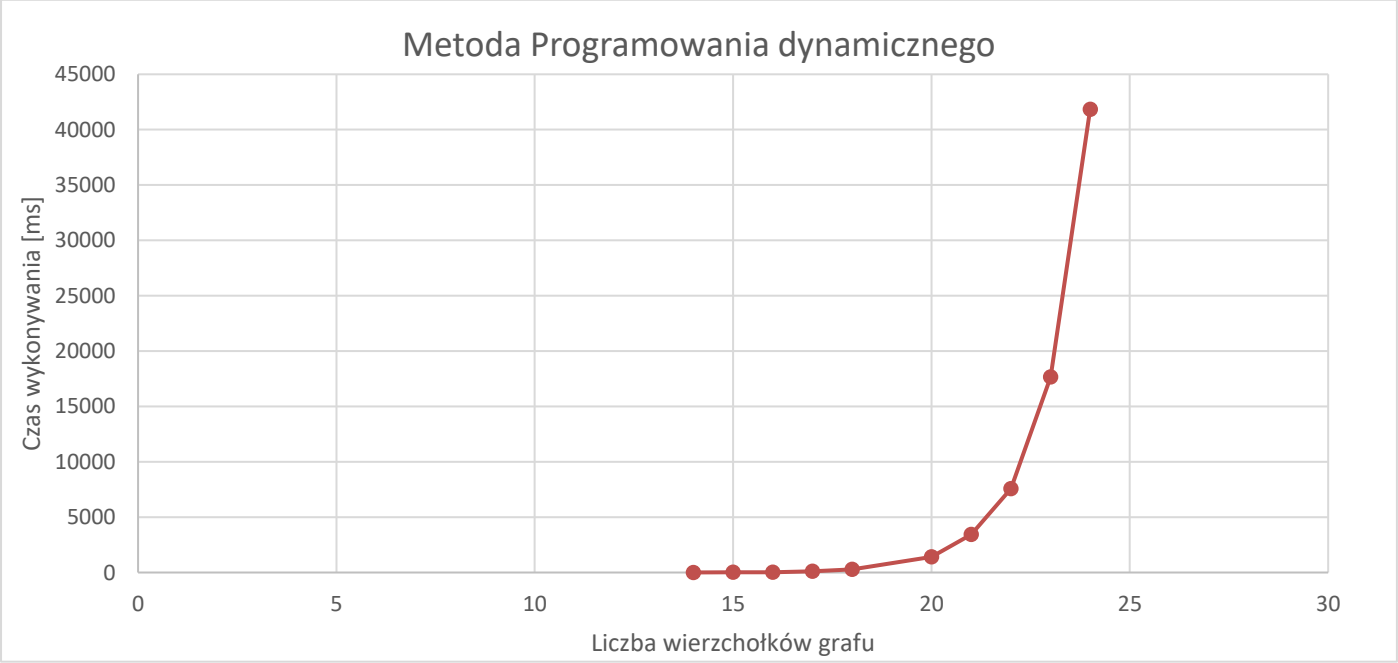
Najkrótsza droga: **0 -> 2 -> 1 -> 0.**

3.2.3. Wyniki pomiarów metody programowania dynamicznego.

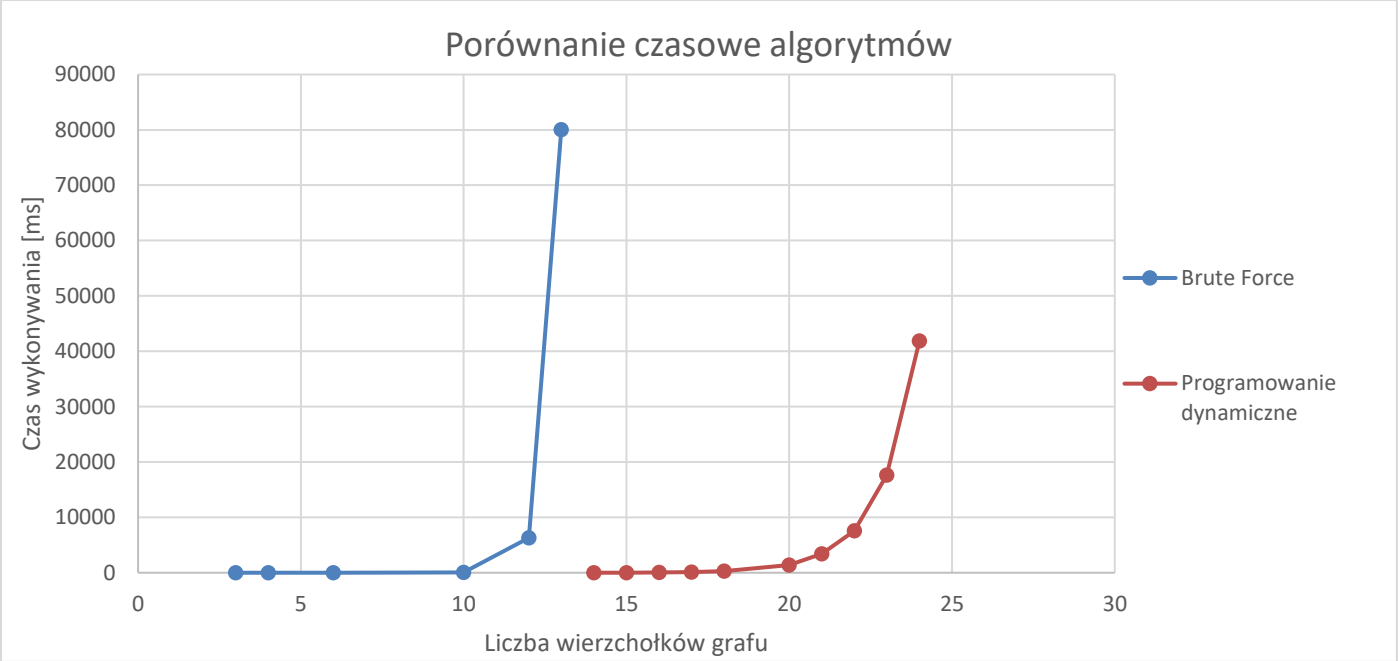
Poniżej zostały przedstawione pomiary czasowe dla programowania dynamicznego:

Tabela 2: Zależności czasowe między rozmiarem grafu, a czasem rozwiązywania TSP przy użyciu programowania dynamicznego.

Nazwa pliku	Liczba wierzchołków grafu	Optymalny koszt cyklu Hamiltona	Uzyskana waga cyklu	Średni czas wykonywania algorytmu w [ms]
tsp_14.txt	14	282	282	6.3360164
tsp_15.txt	15	291	291	14.3460906
data16.txt	16	156	156	35.7130162
br17.atsp	17	39	39	95.507767
data18.txt	18	187	187	274.838689
tsp_20.txt	20	20	20	1415.76021
gr21.tsp	21	2707	2707	3430.76272
tsp_22.txt	22	22	22	7557.962166
tsp_23.txt	23	23	23	17656.1605
gr24.tsp	24	1272	1272	41841.8251



Wykres 2: Zależność czasu wykonywania algorytmu programowania dynamicznego w zależności od rozmiaru macierzy grafu.



Wykres 3: Porównanie algorytmów rozwiązujących problem komiwojażera dla różnej wielkości grafów.

4. Wnioski oraz podsumowanie.

Zadanie projektowe miało na celu zapoznać studenta z różnymi metodami rozwiązania problemu optymalizacyjnego, jakim jest problem komiwojażera. Ze względu na ogromną złożoność obliczeniową rozważanego problemu (zawierającego się w przestrzeni problemów NP-trudnych) niemożliwym było przetestowanie większych instancji problemów w rozsądnym czasie.

Wszystkie algorytmy rozwiązujące problem komiwojażera zostały zaimplementowane w projekcie i działają w pełni poprawnie. Obserwując wyniki czasowe wszystkich algorytmów możemy łatwo zauważyć, że rozwiązanie problemu komiwojażera metodą programowania dynamicznego z użyciem masek bitowych wykonuje się wielokrotnie szybciej od przeglądu zupełnego. Jest to spowodowane naturą samego programowania dynamicznego – rozwiązywanie całego problemu jako szereg mniejszych pod-problemów. Niemniej jednak, redukcja złożoności czasowej programowania dynamicznego względem przeglądu zupełnego pozwoliła wykonać testy dla instancji maksymalnie o **10** wierzchołków większych.