Projektowanie Efektywnych Algorytmów
Projekt

21/10/2022

259193 Kacper Wróblewski

(2) Algorytm Helda-Karpa

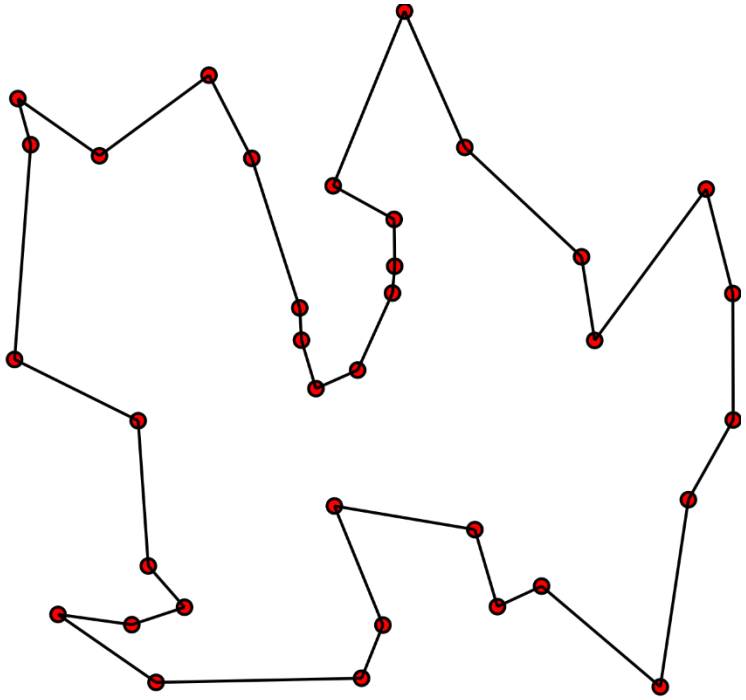
1. Sformułowanie zadania

Zadanie polega na opracowaniu, implementacji i zbadaniu efektywności algorytmu Helda-Karpa rozwiązującego problem komiwojażera w wersji optymalizacyjnej. Algorytm był realizowany na gotowym grafie stworzonym z danych do opracowania.

Należało zbadać zależność czasową od wielkości instancji (jak w zadaniu 1.) oraz pamięciową.

Problem komiwojażera, czyli TSP (*travelling salesman problem*) polega na znalezieniu cyklu Hamiltona w grafie, który ma najmniejszy koszt. Sprowadza się do wyznaczenia najkrótszej ścieżki pomiędzy wierzchołkami przedstawianymi jako miasta, stąd problem podróżującego sprzedawcy. Dana jest określona ilość miast i odległość albo cena podróży pomiędzy nimi i podróżujący musi odwiedzić wszystkie płacąc jak najmniej lub podróżując jak najmniejszą odległość. Jak przedstawiono na Rys. 1, rozwiązaniem jest cykl w grafie zupełnym w postaci listy kolejnych wierzchołków oraz całkowity koszt przebytej drogi.

Problem TSP należy do klasy problemów NP-trudnych, co znaczy, że rozwiązanie nie zawsze jest jasne lub zajmuje zwyczajnie zbyt długo, aby brać je pod uwagę przez co wymagane jest częste zawężanie kryteriów lub kontemplacja jakości algorytmu w danym kontekście.



https://pl.wikipedia.org/wiki/Problem_komiwojażera#/media/Plik:GLPK_solution_of_a_travelling_salesman_problem.svg

Rys. 1. Przykładowe rozwiązanie problemu TSP

2. Metoda

Programowanie dynamiczne *(dynamic programming)* jest to sposób projektowania algorytmów rozwiązujących zagadnienia optymalizacyjne (takie jak *TSP*). Jest to alternatywa dla metod zachłannych. Strategię programowania dynamicznego opracował amerykański matematyk Richard Bellman.

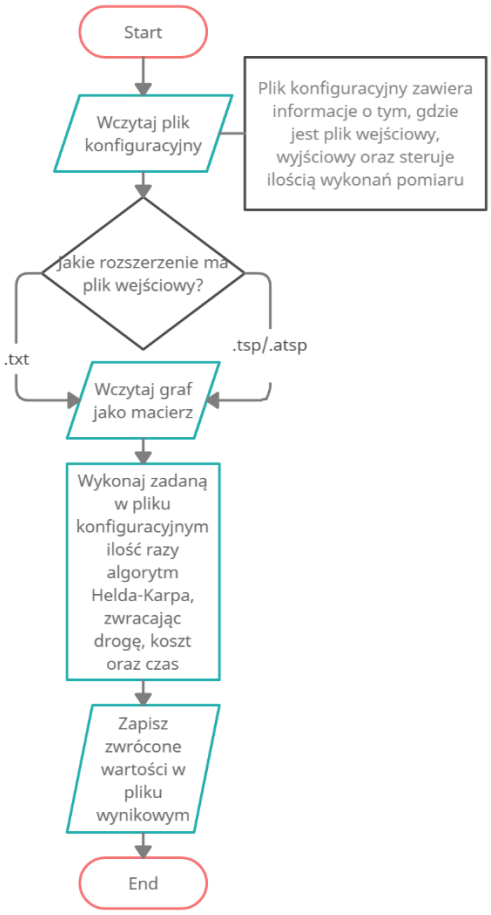
Podstawą tej metody jest podział problemu na mniejsze części, zwane podproblemami, względem kilku parametrów. Przez dużą złożoność pamięciową zastosowanie w programowaniu dynamicznym naiwnych rozwiązań jakim jest np. przegląd zupełny *(ang. brute force)* prowadzi do znacznego wzrostu złożoności pamięciowej problemu.

Kluczem do zaprojektowania algorytmu programowania dynamicznego jest znalezienie funkcji rekurencyjnej, która optymalnie opisuje funkcję celu dzieląc problem na części (zazwyczaj od najmniejszej do największej). Dzięki świadomości, że postępowanie jest optymalne mamy pewność, że kolejne wywołania funkcji również takie będą. W tym podejściu pamiętane są wszystkie poprzednie rozwiązania podproblemu, więc rozbijanie zadania na części nie tylko ma sens w kwestii przydatności, ale również optymalności algorytmów programowania dynamicznego.

Algorytm Helda-Karpa (czasami określany również jako algorytm Bellmana-Helda-Karpa) jest to algorytm oparty na programowaniu dynamicznym służący do rozwiązywania problemu komiwojażera. Jego złożoność czasowa to $O(n^2 2^n)$ a pamięciowa $O(n2^n)$. Pomimo tego, że te wartości są gorsze od najczęściej spotykanych wielomianowych, jest to złożoność lepsza od $O(n!)$ prezentowanej przez przegląd zupełny.

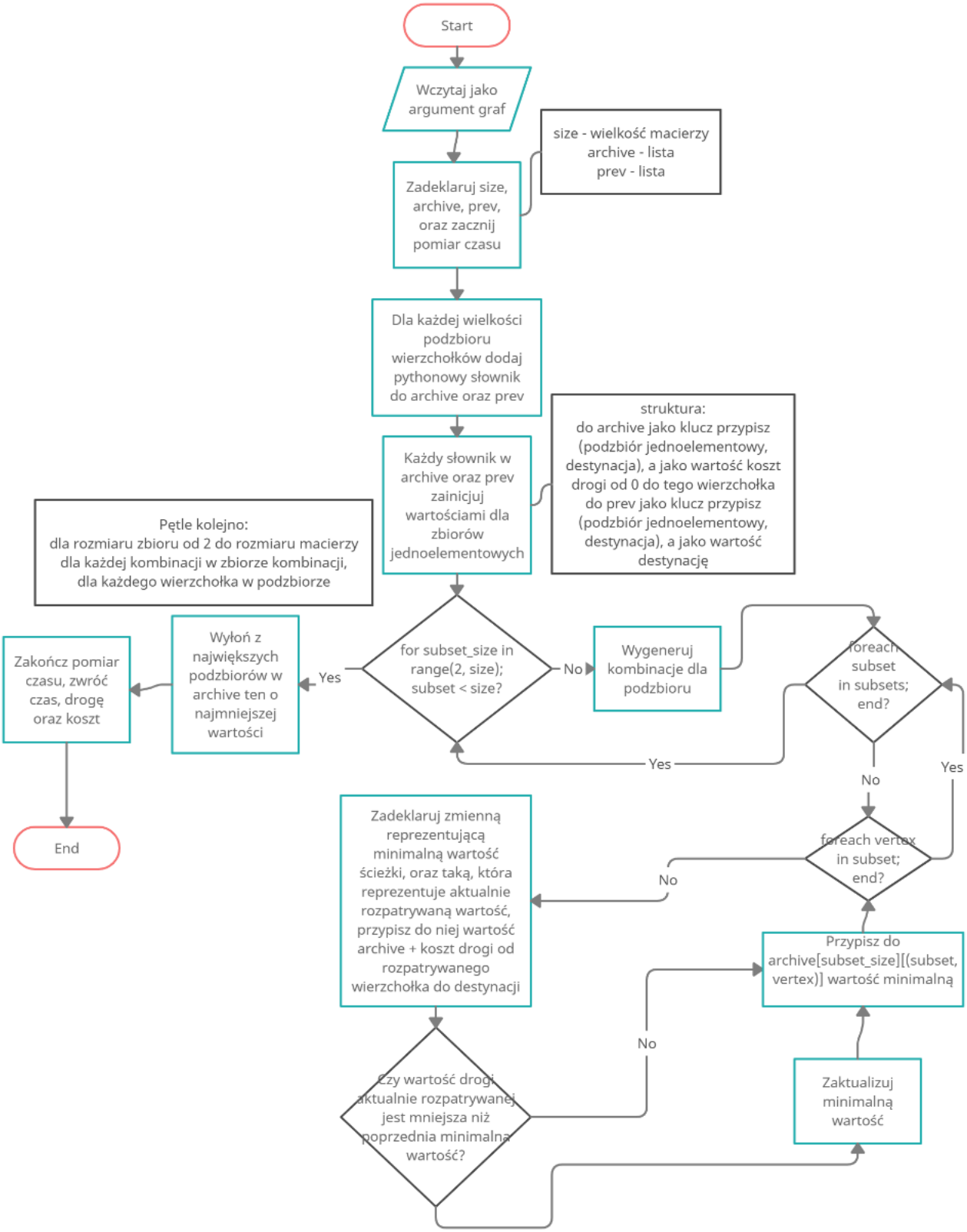
W skrócie, metoda polega na przeglądzie wszystkich podzbiorów (kombinacji) zbioru wierzchołków w rosnącej kolejności wynikowo przypisując kolejne optymalne wartości do kolekcji celem ich zapamiętania. Szczegóły implementacji zaprezentowano w punkcie 3.

3. Algorytm



Rys. 2. Schemat blokowy całego programu

Program, jak przedstawiono na Rys. 2., rozpoczyna się od wczytania danych zawartych w pliku konfiguracyjnym. Potem należy rozpatrzeć przypadek rozszerzenia pliku, gdyż część z badanych zbiorów ma inny format niż *.txt*, który wymaga nieco innej metody odczytu (pliki *.tsp/.astp* można łatwo odczytać za pomocą biblioteki *tsplib95* w pythonie). Następnie program wchodzi w pętlę realizującą zaimplementowany algorytm. Liczba iteracji tej pętli określona jest w pliku konfiguracyjnym. Gdy program zakończy iterację, zapisuje zwrócone dane do pliku, którego ścieżka i nazwa również znajdują się w pliku sterującym, czym powinien zakończyć się program.



Rys. 3. Schemat blokowy implementacji algorytmu Helda-Karpa

Implementacja algorytmu Helda-Karpa rozpoczyna się startem pomiaru czasu.

Następnie deklarowane są wszystkie potrzebne zmienne oraz kolekcje do zapamiętania wyników podproblemów. Potem następuje inicjalizacja list *archive* oraz *prev* wartościami odległości oraz destynacji zbiorów jednoelementowych celem zapamiętania wyników dla każdego podproblemu (wyniki zapisywane są w słowniku, ostatecznie *archive* oraz *prev* utworzą listy słowników).

Kolejną operacją jest utworzenie pętli, która rozpatruje wszystkie rozmiary podzbiorów większych od jeden (pierwszy blok decyzyjny na rys. 3. oraz druga pętla *for* w pseudokodzie na rys. 4. poniżej) oraz wytworzenie listy kombinacji o zadanej wielkości z rozpatrywanego podzbioru. Na tych podzbiorach wywoływana będzie pętla *foreach* badająca każdy podzbiór z listy podzbiorów o aktualnie rozpatrywanej wielkości (trzecia pętla *for* na rys. 4. oraz drugi blok decyzyjny na schemacie blokowym na rys. 3.). Następnie badane są wszystkie wierzchołki *vertex* w badanym podzbiorze pod względem ich sumarycznej wartości drogi z mniejszym o jeden podzbiorem, do których owy *vertex* nie należy (reprezentuje to blok procesu wychodzący przy zdarzeniu *no* z bloku decyzyjnego opisującego pętlę *foreach* dla wierzchołków podzbioru). Można to porównać do wyciągania minimum z wartości drogi dla zbiorów *k*-elementowych w najbardziej zagnieżdżonej pętli w pseudokodzie (rys. 4.).

Po wyjściu ze wszystkich pętli obliczana jest optymalna droga poprzez wyciągnięcie minimalnej wartości drogi ze słowników opisujących największe podzbiory zbioru wierzchołków oraz dodanie do niej kosztu powrotu do wierzchołka startowego.

Następuje zatrzymanie pomiaru czasu.

Na koniec funkcja zwraca listę wierzchołków tworzącą optymalną ścieżkę, jej wartość oraz czas realizacji wszystkich operacji.

```
function algorithm TSP (G, n) is
    for k := 2 to n do
        g({k}, k) := d(1, k)
    end for

    for s := 2 to n-1 do
        for all S ⊆ {2, ..., n}, |S| = s do
            for all k ∈ S do
                g(S, k) := min_{m≠k,m∈S} [g(S\{k}, m) + d(m, k)]
            end for
        end for
    end for

    opt := min_{k≠1} [g({2, 3, ..., n}, k) + d(k, 1)]
    return (opt)
end function
```

https://en.wikipedia.org/wiki/Held–Karp_algorithm

Rys. 4. Pseudokod opisujący implementowany algorytm Helda-Karpa

4. Dane testowe

Do sprawdzenia poprawności działania algorytmu i przeprowadzenia badań wybrano następujący zestaw instancji (wyniki zebrano w punkcie 6.):

http://jaroslaw.mierzwa.staff.iiar.pwr.wroc.pl/pea-stud/tsp/

Plik:	*Optymalne wartości (ścieżka, koszt):*
1. tsp_6_1.txt	*[0, 1, 2, 3, 4, 5], 132*
2. tsp_6_2.txt	*[0, 5, 1, 2, 3, 4], 80*
3. tsp_10.txt	*[0, 3, 4, 2, 8, 7, 6, 9, 1, 5, 0], 212*
4. tsp_12.txt	*[0, 1, 8, 4, 6, 2, 11, 9, 7, 5, 3, 10, 0], 264*
5. tsp_13.txt	*[0, 10, 3, 5, 7, 9, 11, 2, 6, 4, 8, 1, 12, 0], 269*
6. tsp_14.txt	*[0, 10, 3, 5, 7, 9, 13, 11, 2, 6, 4, 8, 1, 12, 0], 282*
7. tsp_15.txt	*[0, 10, 3, 5, 7, 9, 13, 11, 2, 6, 4, 8, 14, 1, 12, 0], 291*

Do programu został dołączony plik *config.ini*, aby sterować parametrami programu w sposób zastępujący:

Plik wejściowy:	
./input/tsp_6_1.txt	*//tutaj zawarta nazwa pliku wywoła odpowiedni graf w programie.*
Plik wyjściowy:	
./output/data.csv	*//ścieżka pliku wyjściowego*
Ilość powtórzeń:	
100	*//ilość razy, z jaką wykona się algorytm*

5. Procedura badawcza

Należało zbadać zależność czasu rozwiązania problemu i złożoność pamięciową od wielkości instancji dla metody Helda-Karpa. W przypadku tego algorytmu w przestrzeni rozwiązań dopuszczalnych nie występowały parametry programu, które mogły mieć wpływ na czas i jakość uzyskanego wyniku. W związku z tym procedura badawcza polegała na uruchomieniu programu sterowanego plikiem inicjującym *.INI*, którego struktura została opisana wyżej.

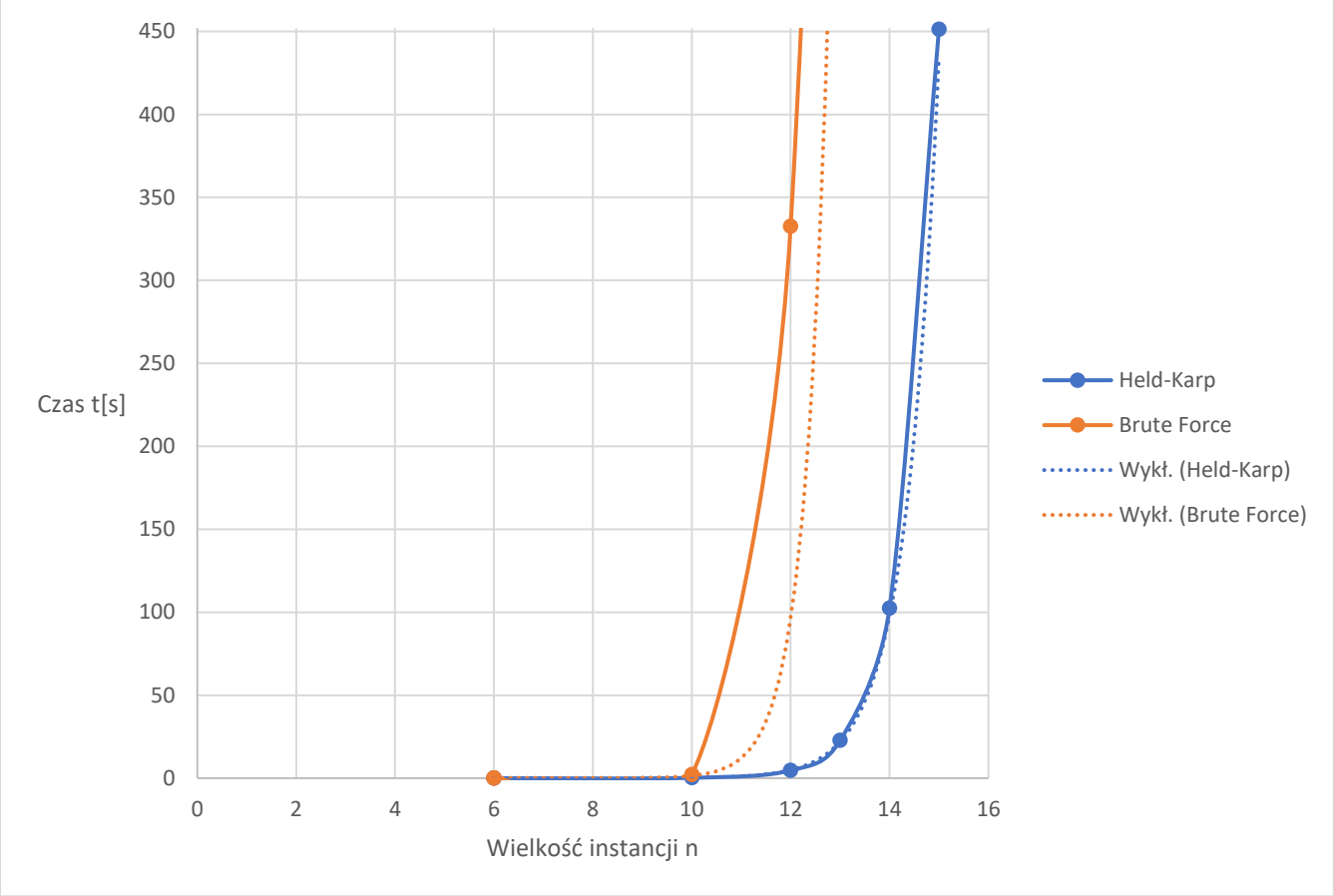
Dla problemów o mniejszych rozmiarach można było wykonać algorytm sto razy na jedno uruchomienie programu. Przy wielkości instancji rzędu trzynaście i więcej wykonanie algorytmu sto razy nie miało większego sensu gdyż rozwiązanie zajmowało zbyt dużą ilość czasu. Zmniejszono zatem ilość powtórzeń wywołania algorytmu do dziesięciu, aby uzyskać jakiekolwiek wyniki.

Wyniki zostały zgromadzone w pliku wyjściowym *dane.csv* znajdującym się w katalogu *output*. Brane pod uwagę było optymalne rozwiązanie problemu oraz średni czas wykonania w sekundach.

Wyniki opracowane zostały w programie MS Excel.

6. Wyniki

Wyniki zgromadzone zostały w pliku *dane.csv* w katalogu *output*. Przedstawione zostały w postaci wykresu zależności czasu uzyskania rozwiązania problemu od wielkości instancji (wykres 1.), dane pomiarowe dla programowania dynamicznego znajdują się w tabeli 1. Wykres 1. porównuje wyniki algorytmu przeglądu zupełnego, jego linię trendu oraz zależności algorytmu Helda-Karpa i jego linię trendu celem porównania czasu działania obu metod.



Wykres 1. Wykres wynikowy dla metody programowania dynamicznego z zadania 2 wraz z porównaniem wyników zadania 1..

rozmiar instancji	*średni czas[s]*
6	*0,0004*
6	*0,0011*
10	*0,2109*
12	*4,6807*
13	*22,6935*
14	*102,3759*
15	*451,2849*

Tabela 1. Wyniki pomiarów dla algorytmu Helda-Karpa

7. Analiza wyników i wnioski

Algorytm Helda-Karpa posiada teoretycznie lepszą złożoność czasową $(O(n^2 2^n))$ niż metoda przeglądu zupełnego $(O(n!))$, co ciekawe jednak, dla instancji wielkości sześć dla obu metod wyniki wychodziły zbliżone lub setną sekundy na korzyść przeglądu zupełnego. Może to mieć związek z nieoptymalnym zaimplementowaniem algorytmu lub zapisem wyników. Niewykluczone jednak, że tak wynika ze złożoności i dla mniejszych instancji *brute force* jest lepszym rozwiązaniem.

Po nałożeniu linii estymacji miedzy punktami pomiarowymi zarówno metoda przeglądu zupełnego jak i programowanie dynamiczne prezentują funkcję wykładniczą. Widoczne jest jednak, że linia *brute force* rośnie znacznie szybciej, co udowadnia jego gorszą złożoność.

Dzięki lepszej złożoności, algorytm programowania dynamicznego pozwala rozwiązywać szybciej większe problemy. W badaniu metody *brute force* nie udało się przekroczyć bariery rozmiaru instancji trzynaście. W programowaniu dynamicznym, dla mniejszych instancji problem rozwiązywany jest nawet sto razy szybciej (dla rozmiaru dwanaście), co potwierdza lepszą strukturę algorytmu.

Dla instancji przekraczających siedemnaście jednak, czas badania wykracza poza zakres założony dla jednego wywołania lub wyczerpuje pamięć, nie wzięto tych wyników zatem pod uwagę.