Projektowanie Efektywnych Algorytmów

Projekt
15.11.2022

254307 Paul Paczyński

(2) Algorytm Held'a-Karp'a

1. Sformułowanie zadania

Zadanie polega na opracowaniu, implementacji i zbadaniu efektywności algorytmu Helda-Karpa rozwiązującego problem komiwojażera. Należy także znaleźć „górną granicę” algorytmu, czyli instancję dla której wykonanie zadania będzie trwało zbyt długo, lub zabraknie pamięci.

2. Metoda

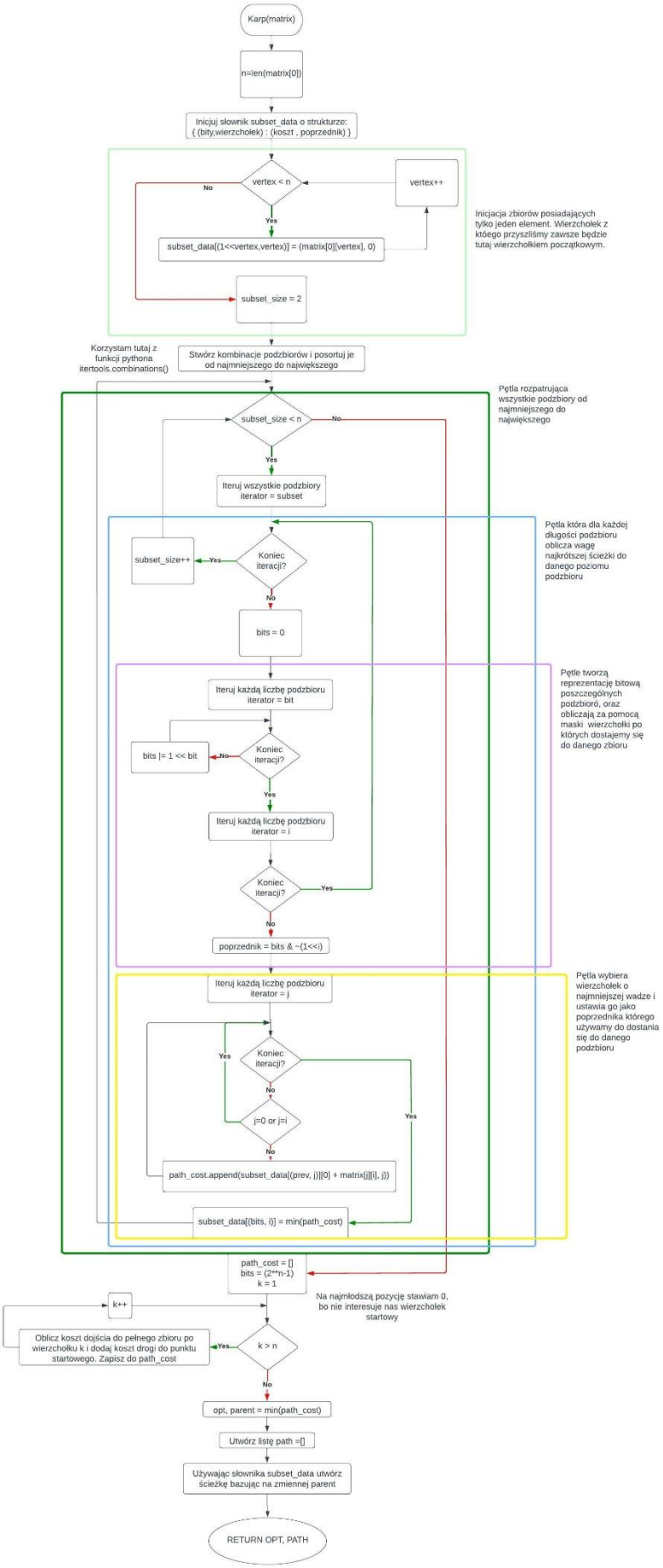
Programowanie dynamiczne jest techniką projektowania algorytmów, która opiera się na podzielenie problemu na mniejsze pod problemy. Od metody dziel i zwyciężaj odróżnia ją to, że pod problemy w metodzie programowania dynamicznego nie są rozłączne. Algorytm oparty na danej metodzie rozwiązuje każdy pod problem tylko raz, zapamiętując jego wynik w odpowiedniej strukturze. Takie dane mogą być potem użyte do rozwiązywania kolejny pod problemów, a dodatkowo dzięki temu unikamy możliwości powtórzenia niepotrzebnych obliczeń.

3. Algorytm

W moim algorytmie główną strukturą którą używałem był słownik z języka Python. W słowniku subset_data przechowywałem dane w postaci (**bity**, **wierzchołek**) : (**koszt**, **poprzednik**), gdzie nawiasy przedstawiają niezmienne tablice (tuple w języku Python). Zmienna **bity** to przedstawienie danego zbioru w postaci bitów, gdzie aby przedstawić obecność danego wierzchołka w zbiorze, należy ustawić „1” na indeksie odpowiadającym wartości wierzchołka. **Wierzchołek** jest rozpatrywanym wierzchołkiem po którym uda nam się dojść do zbioru. **Koszt** to koszt dojścia do zbioru, a **poprzednik** to wierzchołek po którym będziemy mogli odszukać drogę jaką przeszliśmy, wyszukując w tablicy po (bity, poprzednik).

Dzięki tej strukturze danych program jest w stanie wytworzyć i posortować względem długości podzbioru dane, które potem z pomocą operacji bitowych utworzą listę kosztów dotarcia do poszczególnych zbiorów względem danego wierzchołka.

Po obliczeniu optymalnego kosztu największego zbioru program używa zmiennej poprzednika, aby odtworzyć przebytą drogę.


Karp(matrix)
n=len(matrix[0])
Inicjuj słownik subset_data o strukturze:
{ (bity,wierzchołek) : (koszt , poprzednik) }
vertex < n
No
Yes
vertex++
subset_data[(1<<vertex,vertex)] = (matrix[0][vertex], 0)
subset_size = 2
Inicjacja zbiorów posiadających tylko jeden element. Wierzchołek z któego przyszliśmy zawsze będzie tutaj wierzchołkiem początkowym.
Korzystam tutaj z funkcji pythona itertools.combinations()
Stwórz kombinacje podzbiorów i posortuj je od najmniejszego do największego
subset_size < n
No
Yes
Pętla rozpatrująca wszystkie podzbiory od najmniejszego do największego
Iteruj wszystkie podzbiory
iterator = subset
subset_size++
Yes
Koniec iteracji?
No
Pętla która dla każdej długości podzbioru oblicza wagę najkrótszej ścieżki do danego poziomu podzbioru
bits = 0
Iteruj każdą liczbę podzbioru
iterator = bit
bits |= 1 << bit
No
Koniec iteracji?
Yes
Iteruj każdą liczbę podzbioru
iterator = i
Koniec iteracji?
Yes
No
poprzednik = bits & ~(1<<i)
Pętle tworzą reprezentację bitową poszczególnych podzbiorów, oraz obliczają za pomocą maski wierzchołki po których dostajemy się do danego zbioru
Iteruj każdą liczbę podzbioru
iterator = j
Yes
Koniec iteracji?
No
j=0 or j=i
Yes
No
path_cost.append(subset_data[(prev, j)][0] + matrix[j][i], j))
subset_data[(bits, i)] = min(path_cost)
Pętla wybiera wierzchołek o najmniejszej wadze i ustawia go jako poprzednika którego używamy do dostania się do danego podzbioru
path_cost = []
bits = (2**n-1)
k = 1
Na najmłodszą pozycję stawiam 0, bo nie interesuje nas wierzchołek startowy
k++
Oblicz koszt dojścia do pełnego zbioru po wierzchołku k i dodaj koszt drogi do punktu startowego. Zapisz do path_cost
Yes
k > n
No
opt, parent = min(path_cost)
Utwórz listę path =[]
Używając słownika subset_data utwórz ścieżkę bazując na zmiennej parent
RETURN OPT, PATH

4. Dane testowe

Do sprawdzenia poprawności działania algorytmu, wybrano następujący zestaw instancji:

1. tsp_6_1.txt
2. tsp_10.txt
3. tsp_12.txt

Do przeprowadzenia badań użyto następujących zestawów:

1. tsp_6_1.txt
2. tsp_10.txt
3. tsp_12.txt
4. tsp_14.txt
5. tsp_15.txt
6. tsp_17.txt http://jaroslaw.mierzwa.staff.iiar.pwr.wroc.pl/pea-stud/tsp/
7. gr21.tsp http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/gr21.tsp
8. ftv33.atsp http://elib.zib.de/pub/mp-testdata/tsp/tsplib/atsp/ftv33.atsp

5. Procedura badawcza

Należało zbadać zależność czasu rozwiązania problemu od wielkości instancji. Nie występowały tutaj żadne parametry które miałyby wpływ na czas wykonania algorytmu. Procedura badawcza polegała na uruchomieniu programu sterowanego plikiem inicjującym .INI.

Format pliku INI:
<Liczba plików tsp do przetworzenia>
<Plik x> <liczba wykonań>
<Plik x+1> <liczba wykonań>
<Nazwa pliku output.csv>

Treść pliku INI:
8
6_1.txt 10
10.txt 10
12.txt 10
13.txt 10
14.txt 10
15.txt 10
17.txt 10
21.txt 1
wyniki.csv

Każda instancja była wykonywana tyle razy ile podane dla niej było w pliku INI. Do pliku wyjściowego wyniki.csv zapisywane były dane w formacie:

Nazwa pliku; Czas[s];Ścieżka;Koszt

6. Wyniki

6.1 Sprzęt użyty do badania

Procesor: Intel(R) Core(TM) i5-4690K CPU @ 3.50GHz 3.50 GHz
RAM: 8 GB
System: Windows 64bit, procesor x64

6.2 Sposób pomiaru czasu

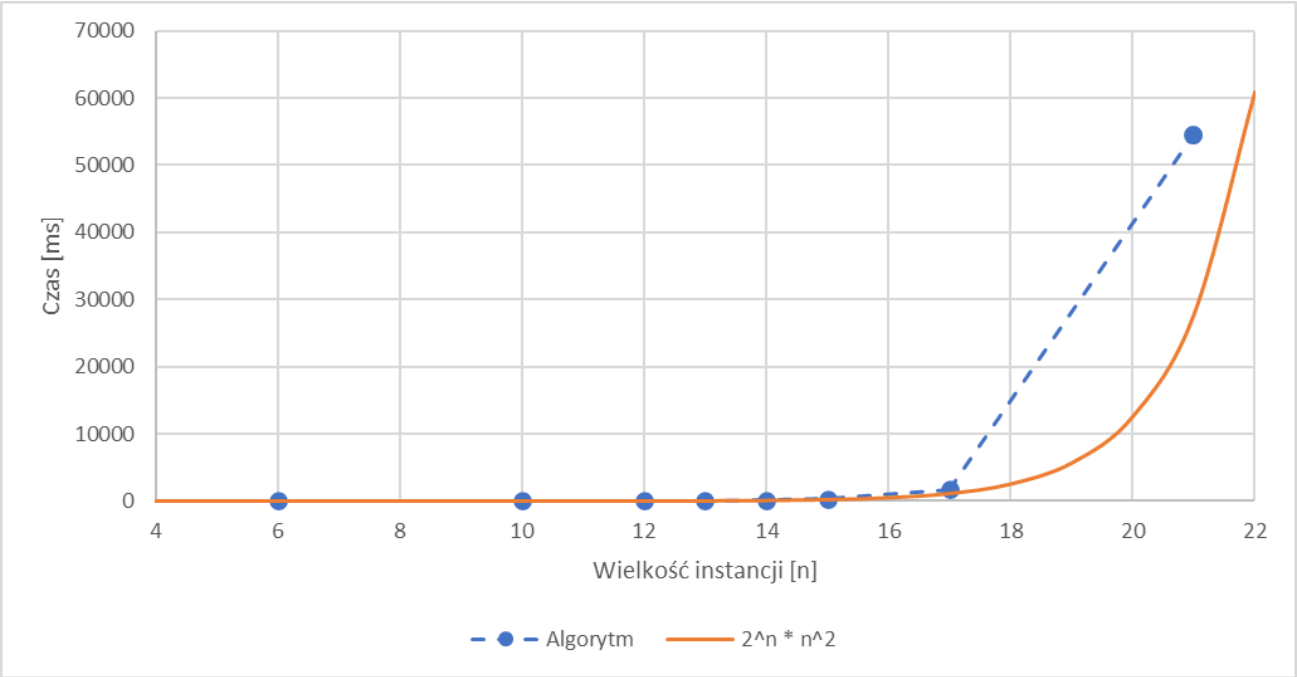
Do pomiaru czasu wykorzystuję bibliotekę time dla języka python. Tworzone są dwie zmienne, start_time i end_time, które przyjmują wartość zwróconą z funkcji time.time(), zwracającej stan zegara. Umiejscowienie algorytmu pomiędzy dwoma zmiennymi, a następnie odjęciu start_time od end_time daje nam czas wykonania algorytmu w sekundach. W przedstawionych wynikach manualnie zamieniłem sekundy na milisekundy dla lepszej czytelności.

6.3 Prezentacja wyników

Wyniki zostały przedstawione w Tabeli 1, oraz pokazane w zestawieniu z funkcją $2^n * n^2$
na Rysunku 1

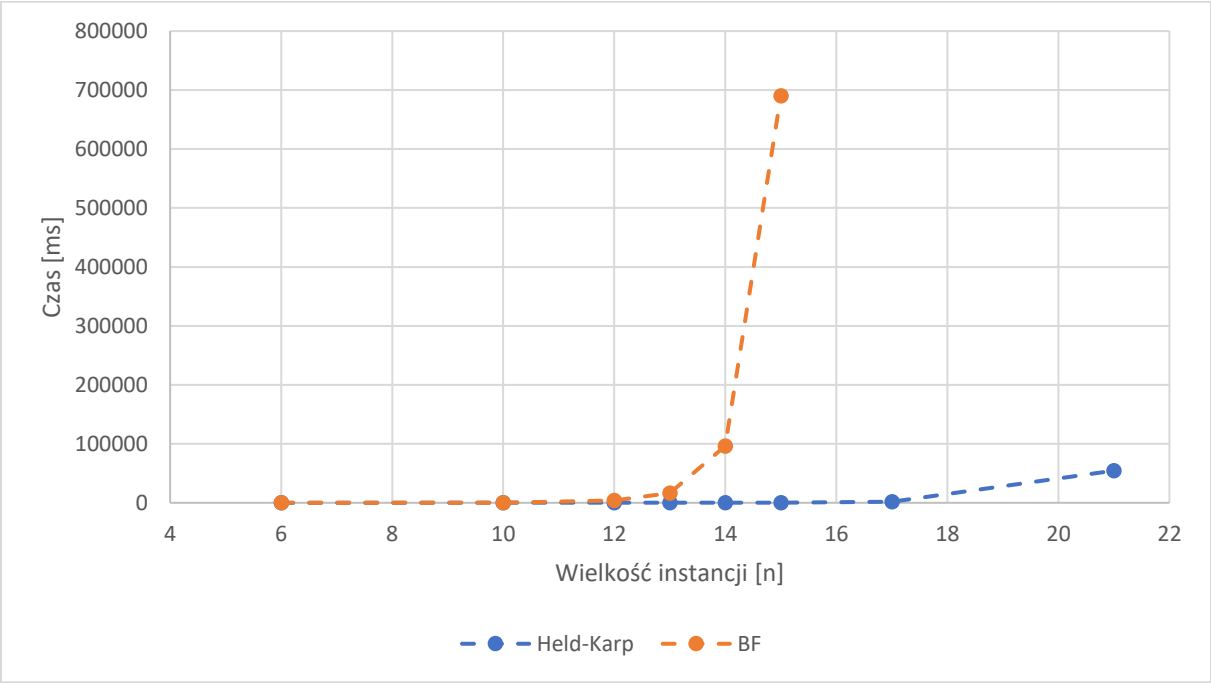
Tabela 1: Wyniki pomiaru czasu działania algorytmu

Wielkość instancji	Optymalna ścieżka	Czas[ms]
6	[0, 1, 2, 3, 4, 5, 0]	0,1002
10	[0, 5, 1, 9, 6, 7, 8, 2, 4, 3, 0]	4,0637
12	[0, 10, 3, 5, 7, 9, 11, 2, 6, 4, 8, 1, 0]	24,2523
13	[0, 12, 1, 8, 4, 6, 2, 11, 9, 7, 5, 3, 10, 0]	58,2749
14	[0, 12, 1, 8, 4, 6, 2, 11, 13, 9, 7, 5, 3, 10, 0]	140,7488
15	[0, 12, 1, 14, 8, 4, 6, 2, 11, 13, 9, 7, 5, 3, 10, 0]	323,8925
17	[0, 11, 16, 8, 7, 4, 3, 15, 14, 6, 5, 12, 10, 9, 1, 13, 2, 0]	1731,7540
21	[0, 11, 3, 10, 19, 18, 16, 9, 17, 12, 13, 14, 20, 1, 2, 8, 4, 15, 5, 7, 6, 0]	54462,1253



Rysunek 1: Wykres zależności czasu działania od wielkości instancji

Na rysunku 2 przedstawiłem wyniki działania poprzedniego algorytmu (Brute Force), w zestawieniu z napisanym algorytmem Helda-Karpa



Rysunek 2: Porównanie czasu działania algorytmu Brute Force z algorytmem Helda-Karpa

7. Analiza wyników i wnioski

Krzywa wzrostu czasu (niebieska) na rysunku 1 ma charakter wykładniczy. Nałożenie krzywej f(n)= $2^n * n^2$ (pomarańczowa) potwierdza, że badany algorytm wyznacza rozwiązania dla problemu komiwojażera w czasie $2^n * n^2$, zależnym względem wielkości instancji badanego problemu. Porównując ten algorytm z algorytmem przeglądu zupełnego (Rysunek 2) można zauważyć, że algorytm przeglądu zupełnego ma znacznie większy czas wykonania. W algorytmie do którego wykonania użyto metody programowania dynamicznego należy mieć na uwadze jednak ilość wykorzystanej pamięci przez program. Podczas wykonywania testu ze zwiększeniem wielkości instancji, zużycie RAM rosło proporcjonalnie. Podczas próby obliczenia instancji o wielkości 33, menedżer zadań pokazywał zużycie 8GB pamięci (100%).