Projektowanie Efektywnych Algorytmow

Projekt

16/11/2021

252690 Karolina Nogacka

(2) Programowanie dynamiczne

spis treści	strona

1. **Sformułowanie zadania:**

   Zadanie polega na stworzeniu algorytmu znajdującego rozwiązanie problemu komiwojażera opartego o metodę programowania dynamicznego.

   Problem komiwojażera (TSP - Traveling salesman problem):

   Zakładając, że ktoś (np. kurier, sprzedawca, listonosz) ma zbiór miast/domów do odwiedzenia szukamy drogi, która zawierać będzie wszystkie wyżej wymienione miejsca i będzie drogą najkrótszą (aby listonosz nie musiał się nachodzić).

   Innymi słowy problem ten jest zagadnieniem optymalizacyjnym , polegającym na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Wszystkich cykli w takim grafie będzie *n*! (w związku z czym złożoność dla większych grafów będzie szybko rosła), pewne z nich będą w rzeczywistości tymi samymi cyklami, zaczynającymi się jednak w różnych węzłach grafu.

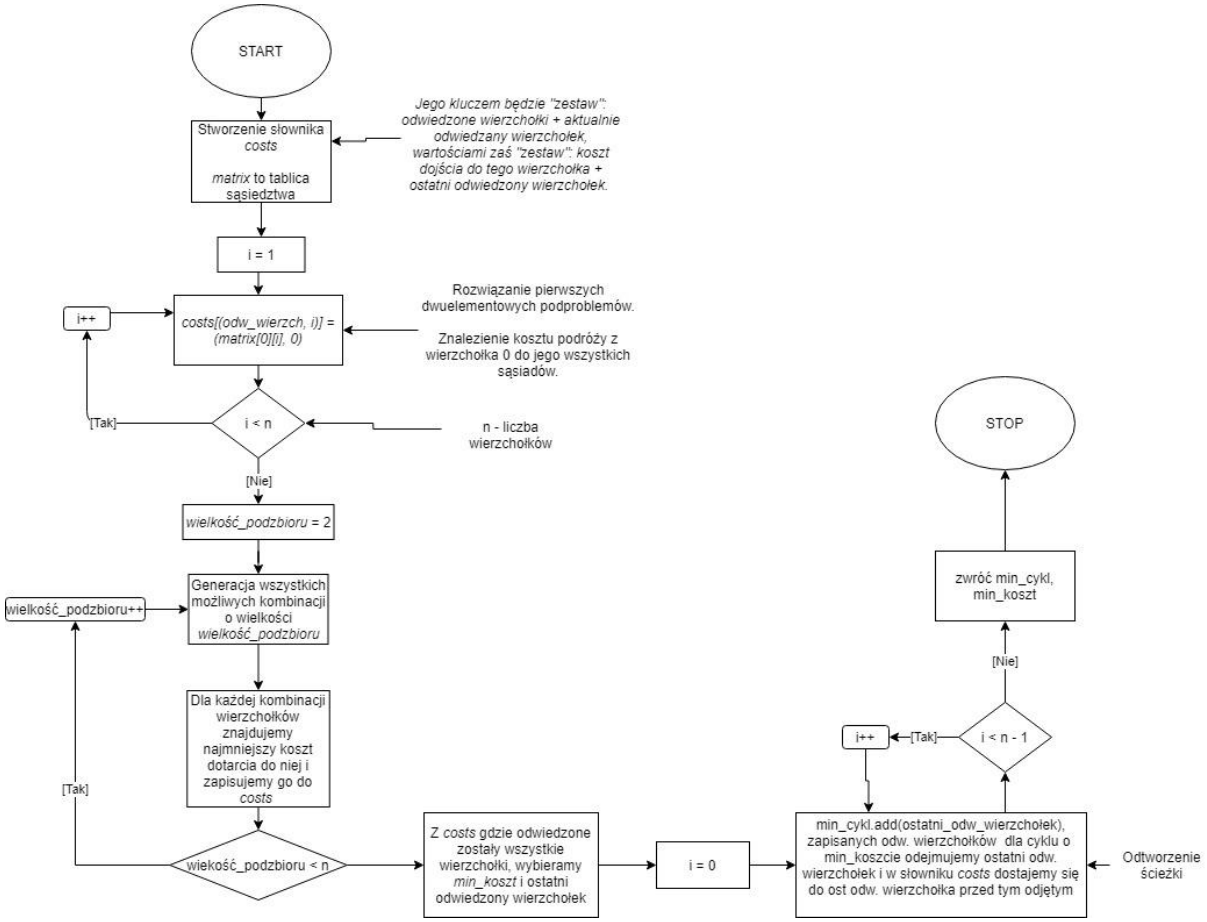
2. Metoda:

Programowanie dynamiczne, polega na rozwiązaniu danego problemu poprzez rozwiązanie mniejszych podproblemów. Metodę tą można stosować, gdy podproblemy dają rozwiązać się w analogiczny do siebie sposób. Rozwiązania podproblemów zapamiętujemy, aby móc je wykorzystać w kolejnych etapach rozwiązywania problemu (przy rozwiązywaniu coraz większych podproblemów).

Oczekujemy złożoności czasowej algorytmu opartego o tą metodę na poziomie **O($n^2 2^n$)**, a pamięciowej **O($n2^n$).**

3. Algorytmy:

Dla problemu komiwojażera metoda programowania dynamicznego odnajdzie wszystkie możliwe cykle Hamiltona i wybierze ten o najmniejszym koszcie. W przeciwieństwie do algorytmu opartego na przeglądzie zupełnym, algorytm oparty o programowanie dynamiczne nie będzie dla każdego cyklu liczył jego kosztu zupełnie od podstaw. Algorytm skorzysta z obliczonych wcześniej kosztów dla elementów cyklu (podproblemy).



Rysunek 1. Schemat blokowy algorytmu znajdującego rozwiązanie problemu komiwojażera, opartego na metodzie programowania dynamicznego.

4. Dane wejściowe i wyjściowe

Co zawiera program badawczy:

- Plik wykonywalny tsp_dp.exe
  Aby działał poprawnie musi być w jednym katalogu z plikiem config.ini i pliki tekstowe z grafami.

- Pliki tsp_6_1.txt, tsp_6_2.txt, tsp_10.txt, tsp_12.txt, itd.
  Zawierają grafy w postaci macierzy sąsiedztwa, które będziemy badać. Źródło: http://jaroslaw.mierzwa.staff.iiar.pwr.wroc.pl/pea-stud/tsp/ [data dostępu: 13.11.2021, 17:48]

- Pliki gr21.tsp, ulysses22.tsp, br17.atsp, itd.
  Zawierają grafy w postaci macierzy sąsiedztwa, które będziemy badać. Źródło: http://jaroslaw.rudy.staff.iiar.pwr.wroc.pl/pea.php [data dostępu: 13.11.2021, 17:54]

- Plik inicjujący config.ini

```ini
[data]
tsp_6_1 = tsp_6_1.txt
tsp_6_2 = tsp_6_2.txt
tsp_10 = tsp_10.txt
tsp_12 = tsp_12.txt
tsp_13 = tsp_13.txt
tsp_14 = tsp_14.txt
tsp_15 = tsp_15.txt
tsp_17 = tsp_17.txt
tsp_6_1_opti = 132; [0, 1, 2, 3, 4, 5, 0]
tsp_6_2_opti = 80; [0, 5, 1, 2, 3, 4, 0]
tsp_10_opti = 212; [0, 3, 4, 2, 8, 7, 6, 9, 1, 5, 0]
tsp_12_opti = 264; [0, 1, 8, 4, 6, 2, 11, 9, 7, 5, 3, 10, 0]
tsp_13_opti = 269; [0, 10, 3, 5, 7, 9, 11, 2, 6, 4, 8, 1, 12, 0]
tsp_14_opti = 282; [0, 10, 3, 5, 7, 9, 13, 11, 2, 6, 4, 8, 1, 12, 0]
tsp_15_opti = 291; [0, 12, 1, 14, 8, 4, 6, 2, 11, 13, 9, 7, 5, 3, 10, 0]
tsp_17_opti = 39; [0, 11, 13, 2, 9, 10, 1, 12, 15, 14, 5, 6, 3, 4, 7, 8, 16, 0]

gr21 = gr21.tsp
ulysses22 = ulysses22.tsp
gr24 = gr24.tsp
fri26 = fri26.tsp
bays29 = bays29.tsp
;att48 = att48.tsp
;eil51 = eil51.tsp
;berlin52 = berlin52.tsp
br17 = br17.atsp
ftv33 = ftv33.atsp
ftv35 = ftv35.atsp
ftv38 = ftv38.atsp
p43 = p43.atsp

gr21_opti = 2707
ulysses22_opti = 7013
gr24_opti = 1272
fri26_opti = 937
bays29_opti  = 2020
;att48_opti  = 10628
;eil51_opti  = 426
;berlin52_opti  = 7542
br17_opti  = 39
ftv33_opti  = 1286
ftv35_opti  = 1473
ftv38_opti  = 1530
p43_opti  = 5620

[result]
tsp_dp = tsp_dp.csv
tsp_bs = tsp_bs.csv
tsp_bf = tsp_bf.csv
tsp_df = tsp_df.csv
```

Rysunek 2. Zawartość pliku config.ini

Pierwsze zdjęcie zawiera nazwy plików ze strony http://jaroslaw.mierzwa.staff.iiar.pwr.wroc.pl/pea-stud/tsp/, oraz optymalne rozwiązania dla grafów w nich zawartych. Dwa pozostałe zdjęcia to nazwy i optymalne rozwiązania dla grafów ze strony http://jaroslaw.rudy.staff.iiar.pwr.wroc.pl/pea.php oraz nazwy plików wyjściowych.

Sekcja [data] zawiera nazwę plików wejściowych z grafami.
Sekcja [result] zawiera nazwę pliku wyjściowego.

- Skrypty tsp_dp.py, my_writer.py:

Skrypt tsp_dp.py jest głównym plikiem programu, zawiera on wywołania metod z innych plików. my_writer.py zapewnia poprawny zapis do plików wynikowych.

- Plik tsp_df_out-analiza.csv:

Zawiera zbiorcze dane i wykresy.

5. Procedura badawcza

- **Kolejność wykonywanych badań:**
    - Pobieranie danych z pliku inicjującego,
    - Wczytanie grafów z plików wejściowych,
    - Uruchomienie badań dla grafów.

Wynikiem działania programu (zapisywanym do pliku) jest czas znalezienia rozwiązania dla każdej instancji, długość najkrótszego cyklu, oraz sam cykl (w postaci listy węzłów).

Wyniki zostały opracowane po 20-krotnym uruchomieniu programu (algorytm wykonał się 20 razy dla każdej instancji). Jako górne ograniczenie działania programu przyjęto 1 godzinę. Podczas tej godziny program maksymalne zdołał policzyć wynik dla instancji gr24.tsp (25 min trwała praca nad tą jedną instancją).

Z dodatkowych instancji badania zostały wykonane dla: gr21.tsp, ulysses22.tsp, gr24.tsp, br17.atsp.

Program nigdy nie wyczerpał pamięci komputera, jednak jego praca ponad 1 godzinę znacznie zakłócała prace komputera, w związku z czym algorytm został przerwany.

- **Specyfikacja sprzętu:**
    a. Procesor Intel i7-10510U, 1.80GHz – 2.30 GHz
    b. 16,0 GB pamięci ram
    c. System Windows 10 Home Edition

- **Metoda badania zużycia pamięci:**

```
201    def work(name, matrix):
202        print("Liczę dla: " + name)
203        start_time = time.time()
204        path, cost = held_karp2(matrix)
205        end_time = time.time() - start_time
206
207        tsp_result = [name, len(matrix), end_time, memory(), cost, path]
208        # tsp_result = [name, len(matrix), end_time, min(paths_costs), paths[low_cost_index]]
209        return tsp_result
```

Rysunek 2. Metoda pomiaru czasu.

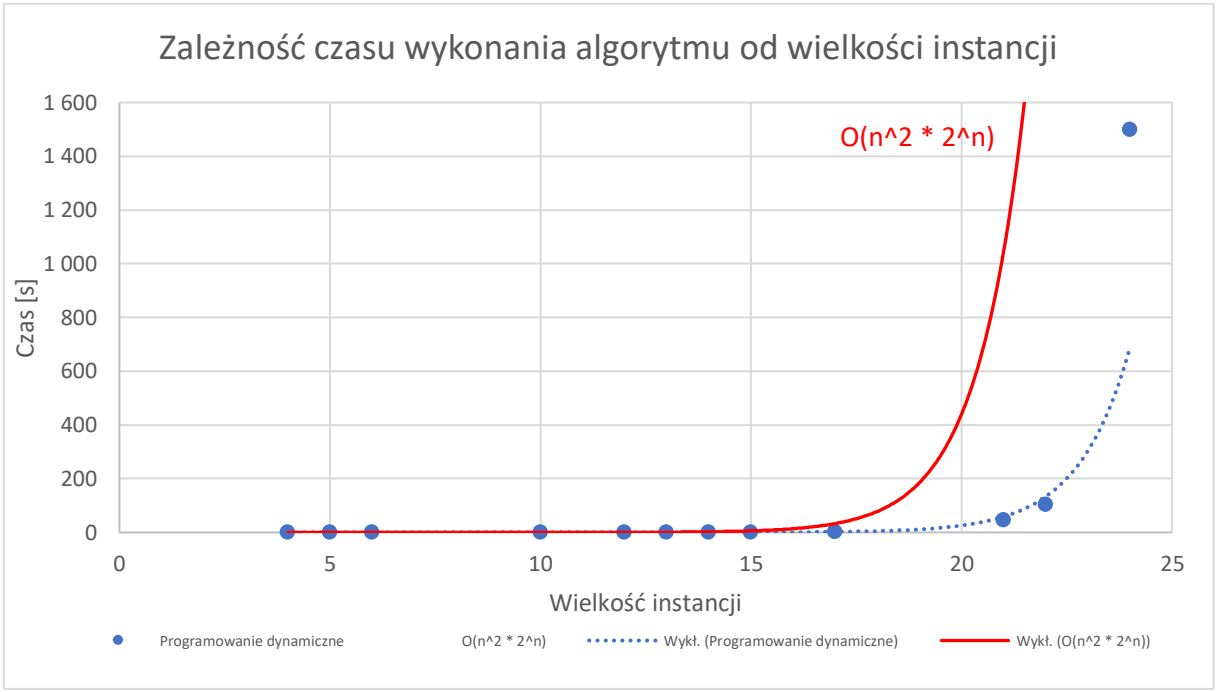
W funkcji *work()* przed i po uruchomieniu funkcji *held_karp2()* mierzony jest czas za pomocą funkcji z biblioteki python'a *time. Różnica zapisywana jest do pliku wyjściowego jako czas wykonania algorytmu dla zadanej instancji.*

- **Metoda badania zużycia pamięci:**

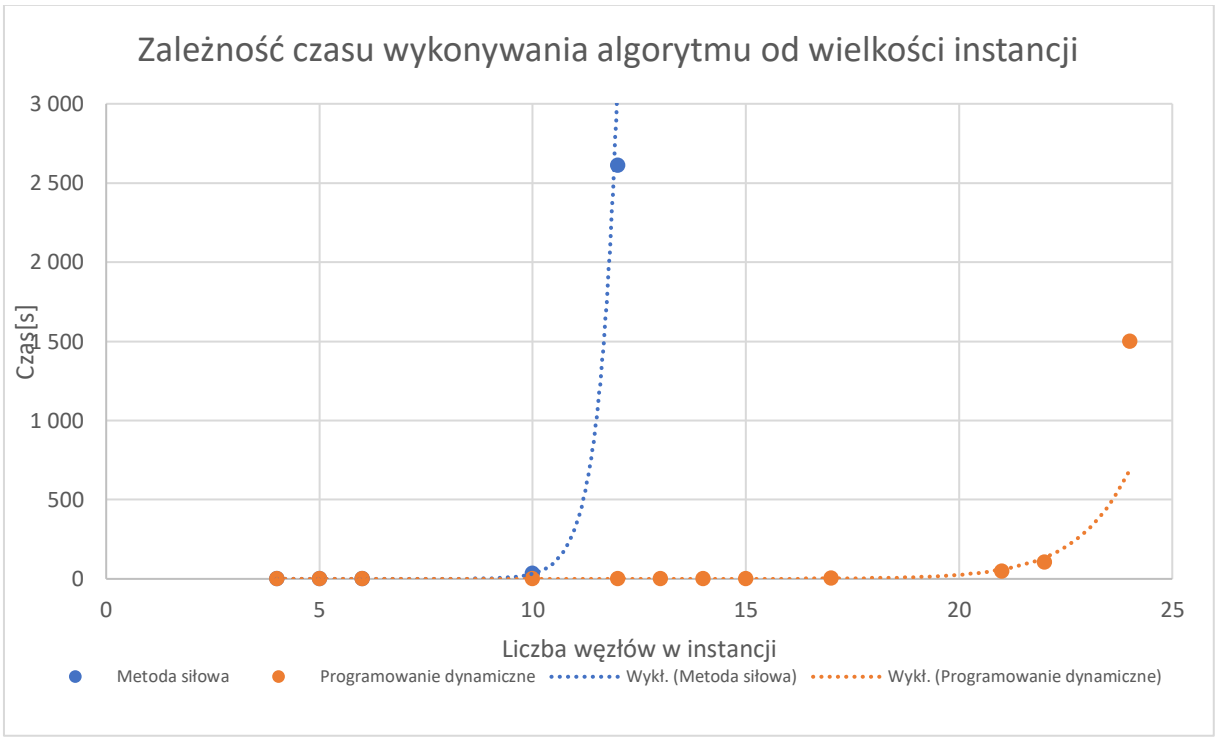
```
14    def memory():
15        w = wmi.WMI('.')
16        result = w.query("SELECT WorkingSet FROM Win32_PerfRawData_PerfProc_Process WHERE IDProcess=%d" % os.getpid())
17        return int(result[0].WorkingSet)
```

Rysunek 3. Metoda badania zużycia pamięci. Sprawdza pamięć zużywaną przez proces (działającą aplikację).

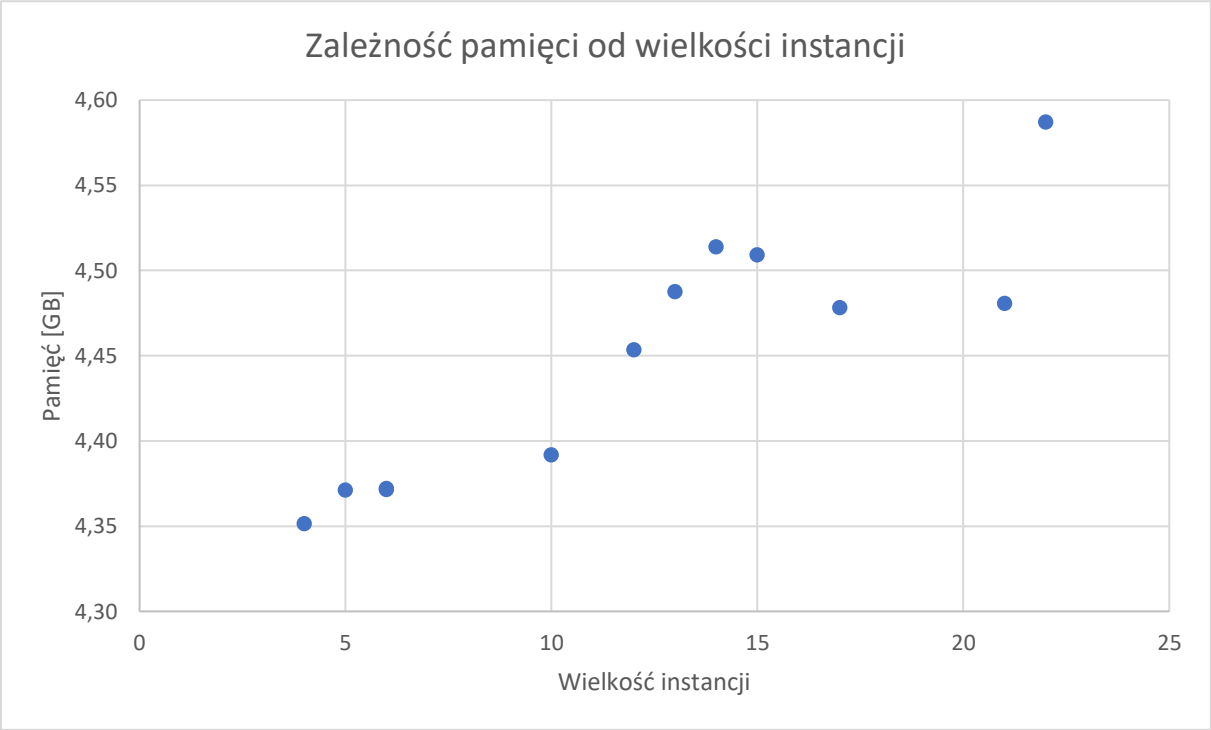
6. Wyniki



Rysunek 4. Zależność czasu pracy algorytmu w zależności od liczby węzłów instancji.



Rysunek 5. Porównanie czasu pracy algorytmu opartego na brute force i dynamic programming.



Rysunek 6. Zużycie pamięci w zależności od wielkości instancji (Programowanie dynamiczne)

7. Analiza wyników i wnioski

Algorytm skonstruowany w oparciu o metodę programowania dynamicznego daje poprawne wyniki dla problemu komiwojażera i ma spodziewaną złożoność czasową **O($n^2 2^n$)** (*Rysunek 4*.)

Złożoność pamięciowa (*Rysunek 6.)* zawiera znaczne fluktuacje na wykresie, co uniemożliwia przeprowadzenie jednoznacznej linii trendu. Jest to najprawdopodobniej spowodowane różnicami w poszczególnych instancjach. Widać jednak trend rosnący dla zużycia pamięci, co zgodne jest z rosnącym rozmiarem instancji.

Tak jak się spodziewano, złożoność czasowa algorytmu opartego o metodę programowania dynamicznego okazała się szybsza, niż algorytmu opartego na metodzie siłowej (*Rysunek 5.*).