Projektowanie efektywnych algorytmów

Projekt

02.11.2020

248821 Michał Rajkowski

(2)Held-Karp

Spis treści

1. Sformułowanie zadania

Zadanie polega na opracowaniu, implementacji i zbadaniu efektywności algorytmu Helda-Karpa rozwiązującego problem komiwojażera w wersji optymalizacyjnej.

Problem komiwojażera (ang. Travelling salesman problem) jest to zagadnienie optymalizacyjne polegające na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym.

Zagadnienie to można podzielić na dwa podtypy ze względu na wagi ścieżek łączących wierzchołki grafu. W symetrycznym problemie komiwojażera (STSP) dla dowolnych dwóch wierzchołków A B krawędź łącząca wierzchołek A z B ma taką samą wagę jak ta łącząca wierzchołek B z A. Natomiast w asymetrycznym problemie komiwojażera (ATSP) wagi te mogą się różnić.

Główną trudnością problemu jest duża liczba danych do analizy. W przypadku problemu symetrycznego dla n miast liczba kombinacji wynosi $\frac{(n-1)!}{2}$.

Dla problemy asymetrycznego jest to aż n!.

2. Metoda

Programowanie dynamiczne jest strategią projektowania algorytmów, stosowaną przeważnie do rozwiązywania zagadnień optymalizacyjnych. Opiera się na podziale rozwiązywanego problemu na podproblemy względem kilku parametrów. Następnie algorytm rozwiązuje podproblemy i zapamiętuje ich wyniki, na których podstawie możliwe jest uzyskanie rozwiązania dla całego problemu.

W przypadku naszego algorytmu programowanie dynamiczne objawia się w podziale problemu na poszczególne stany. Każdy stan określany jest przez dwa parametry, zbiór wierzchołków znajdujących się w aktualnie rozpatrywanej ścieżce S oraz wierzchołek k który jest ostatnim podłączonym wierzchołkiem do ścieżki. Wartość każdego stanu oznacza długość ścieżki o podanych parametrach.

Podział na te konkretne stany wynika z prostego faktu. Każda podścieżka minimalnej ścieżki jest sama w sobie minimalną ścieżką. Zatem możemy korzystać z obliczonych już minimalnych ścieżek aby znajdować kolejne minimalne ścieżki. Nie należy jednak myśleć iż proces ten zachodzi w sposób zachłanny, nie wiemy bowiem czy konkretny wierzchołek będzie bezpośrednio połączony z daną ścieżką, czy należy połączyć go na dalszym etapie działania algorytmu. Z tego powodu rozpatrujemy wszystkie możliwe zbiory wierzchołków i na tej podstawie wybieramy najlepsze dla nich ścieżki.

Rozwiązanie to pozwala na obniżenie złożoności czasowej, bowiem nie musimy przeglądać wszystkich możliwych kombinacji (n!), a jedynie wszystkie możliwe stany których ilość przystaje do liczby elementów zbioru n-elementowego. Dla każdego takiego stanu wykonujemy minimalizacje, które zostaną wykonane w czasie n^2. Zatem całkowity czas pracy algorytmu będzie przystawał do $O(2^n * n^2)$.

Aby przekonać się jak duża jest to optymalizacja możemy skorzystać ze wzoru Stirlinga. Niestety nie odbywa się to bez żadnego dodatkowego kosztu. Przez to, iż algorytm zapamiętuje wartości stanów zużywa on znacznie więcej pamięci niż rozwiązanie zachłanne. Złożoność pamięciowa będzie także przystawała do $O(2^n)$, gdyż liczba przechowywanych stanów przystaje do tej właśnie funkcji.

$$n! \approx \left(\frac{n}{e}\right)^n \sqrt{2\pi n}$$

Wzór Stirlinga

3. Algorytm

3.1 Pseudokod.

Program został utworzony w oparciu o pseudokodu zamieszczonego na anglojęzycznej stronie Wikipedii dotyczącej algorytmu Helda-Karpa.

```
function algorithm TSP (G, n) is
    for k := 2 to n do
        C({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
                C(S, k) := min_m≠k,m∈S [C(S\{k}, m) + d_m,k]
            end for
        end for
    end for

    opt := min_k≠1 [C({2, 3, . . . , n}, k) + d_k, 1]
    return (opt)
end function
```

Źródło: https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm

3.1 Komentarz do zastosowanych rozwiązań

Zadanie mogło zostać wykonane w „leniwy sposób”, dana jest rekurencyjna metoda która oblicza wartość poszczególnych stanów zaczynając od stanu pełnego zbioru wierzchołków i w sposób rekursyjny schodząc w dół. Przy takich zejściach jako parametr przekazywany by był zbiór wierzchołków stanu który chcemy obliczyć. Jest to rozwiązanie bardzo niewydajne pamięciowo, tworzymy bowiem dużą ilość zbiorów na różnych poziomach rekursji.

Chcielibyśmy aby nasze stany miały postać C(S, k), gdzie S jest zbiorem wierzchołków. Możemy dokonać jednak ciekawej obserwacji. S mogłoby przyjmować formę S[][][]..[], gdzie każdy wymiar tablicy oznacza kolejny wierzchołek zbioru, a jego obecność jest symbolizowane przez wpisanie 0 lub 1 w odpowiednią krotkę. Możemy jednak zauważyć że tablica ta w pamięci komputera tak naprawdę przyjmuje wartość tablicy jednowymiarowej, a każdy kolejny wierzchołek i oznacza przejście o 2^i elementów w celu znalezienia interesującej nas komórki tabeli. Z tego wynika kluczowa dla naszego rozwiązania obserwacja:

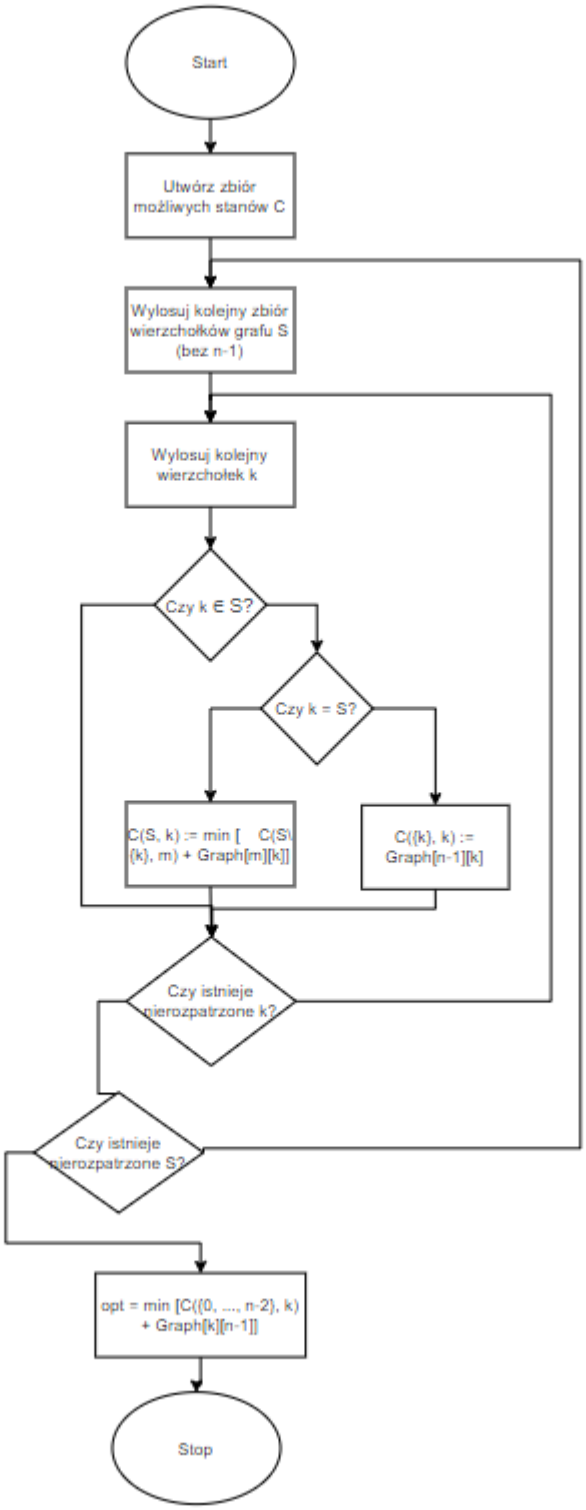
Zbiór {1, 2, 4, 6} może być reprezentowany przez liczbę 2^1 + 2^2 + 2^4 + 2^6 = 1010110_2. Przyporządkowanie to jest jednoznaczne. Nie musimy zatem przekazywać parametrów rekursji w postaci zbioru a jedynie pojedynczej liczby.

Jako iż nasze stany określane są przez liczbę a nie zbiór, a musimy przejść przez wszystkie możliwe stany, można pokusić się o rozwiązanie w sposób iteracyjny. Musimy mieć jednak pewność, że licząc wartość danego stanu nie korzysta on ze stanów jeszcze nieobliczonych. Pewność tą jednak mamy, gdyż:

A + B > A oraz A + B > B , gdy A, B ∈ Z+. A + B jest kodem naszego aktualnego stanu, a A oraz B to stany z których on korzysta. Widzimy zatem iż w rozwiązaniu iteracyjnym zawsze stany pomniejsze są obliczone przed stanem się z nich składającym.

W ten sposób uzyskany został opracowany w zadaniu algorytm iteracyjny.

3.2 Schemat blokowy



3.3 Kod

```
int TSP_HP(){  //metoda na podstawie pseudokodu z wikipedii https://en.w

    //Utwórz zbiór możliwych stanów C
    int** C;
    int n = cnt;

    C = new int*[1<<(n-1)];
    for (int i = 0; i < 1<<(n-1); i++){
        C[i] = new int[n];
        for(int j = 0; j < n; j++){
            C[i][j] = INT_MAX;
        }
    }

    //wylosuj kolejny zbiór wierzchołków grafu S
    for (int S = 1; S < 1<<(n-1); S++){
        //wylosuj kolejny wierzchołek k
        for(int k = 0; k < n-1; k++){
            //Czy k należy do zbioru?
            if(S & 1 << k){
                //czy k jest jedynym wierzchołkiem zbioru?
                if(S == 1 << k){
                    //C({k}, k) := Graph[n-1][k]
                    C[S][k] = tab[n-1][k];
                }else{
                    //C(S, k) := min(C(S\{k}, m) + Graph[m][k])
                    int S_bez_k = S - (1<<k);
                    for(int m = 0; m < n-1; m++){
                        if(S_bez_k & 1 << m){
                                C[S][k] = min(
                                    C[S][k],
                                    C[S_bez_k][m] + tab[m][k]
                                );
                        }

                    }
                }
            }
        }
    }

    //opt = min(C((0, ..., n-2), k) + Graph[k][n-1])
    int opt = INT_MAX;
    for(int k = 0; k < n-1; k++){
        opt = min(
            opt,
            C[(1 << (n-1)) -1 ][k] + tab[k][n-1]
        );
    }

    //opt = min(C((0, ..., n-2), k) + Graph[k][n-1])
    int opt = INT_MAX;
    for(int k = 0; k < n-1; k++){
        opt = min(
            opt,
            C[(1 << (n-1)) -1 ][k] + tab[k][n-1]
        );
    }

    //Metoda otrzymywania ścieżki
    path = new int[n+1];
    path[0] = n-1;
    path[n] = n-1;
    int iterator = n-1;

    int z = n-1;
    int S = (1<< n-1) - 1;
    int min_k, min_path;
    while(S != 0){
        min_k = 0;
        min_path = INT_MAX;
        for(int k = 0; k < n-1; k++){
            if(!(S & (1 << k))){
                continue;
            }
            if(min_path > (C[S][k] + tab[k][z])){
                min_path = (C[S][k] + tab[k][z]);
                min_k = k;
            }
        }
        path[iterator] = min_k;

        z = min_k;
        S = S - (1 << z);
        iterator--;
    }

    //Zwrócenie wyniku opt - optymalna ściezka
    return opt;

}
```

4. Dane testowe

Do sprawdzenia poprawności działania algorytmu oraz do badań wybrano następujący zestaw instancji:

- tsp_17.txt
- tsp_18.txt
- tsp_19.txt
- tsp_20.txt
- tsp_21.txt
- tsp_22.txt
- gr17.txt
- gr21.txt
- gr24.txt
- br17.txt

tsp_17.txt to plik testowy pochodzący ze strony doktora Mierzwy:

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

tsp_18.txt – tsp_22.txt to dane testowe wygenerowane na potrzeby testowania poprawności działania algorytmów.

Pliki gr17.txt – gr24.txt oraz br17.txt zostały utworzone na bazie stron:

http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html

http://elib.zib.de/pub/mp-testdata/tsp/tsplib/atsp/index.html

i zostały okrojone o niepotrzebne nagłówki pliku aby łatwiej było obsługiwać je w programie, stąd rozszerzenie txt.

5. Procedura badawcza

Należało zbadać zależność czasu rozwiązania problemu od wielkości instancji. W przypadku algorytmu realizującego programowanie dynamiczne 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 (format pliku: nazwa_instancji liczba_wykonań rozwiązanie_optymalne [ścieżka optymalna]; nazwa_pliku_wyjściowego).

tsp_17.txt 20 39 16 8 7 4 3 15 14 6 5 12 10 9 1 13 2 11 0 16

tsp_18.txt 20 630 17 5 6 11 1 15 8 9 13 0 3 7 12 4 10 2 14 16 17

tsp_19.txt 20 751 18 8 15 10 11 3 17 1 0 5 6 12 2 7 13 9 16 14 4 18

tsp_20.txt 20 692 19 10 5 15 12 1 6 14 18 16 3 8 11 13 2 17 4 9 7 0 19

tsp_21.txt 20 678 20 15 14 9 17 7 2 6 1 4 0 8 16 5 10 19 3 13 11 12 18 20

tsp_22.txt 20 754 21 3 0 12 6 13 1 17 14 7 15 18 19 10 9 5 16 2 11 8 20 4 21

final_test.csv

Każda instancji rozwiązywana była zgodnie z liczbą jej wykonań, np. tsp_17.txt wykonana została 20 razy. Do pliku wyjściowego final_test.csv zapisywany był czas wykonania, otrzymane rozwiązanie (koszt ścieżki) oraz ścieżka (numery kolejnych węzłów). Plik wyjściowy zapisywany był w formacie csv. Poniżej przedstawiono fragment zawartości pliku wyjściowego:

tsp_17.txt 5 39 16 8 7 4 3 15 14 6 5 12 10 9 1 13 2 11 0 16

108

108

120

108

108

(…)

tsp_18.txt 5 630 17 5 6 11 1 15 8 9 13 0 3 7 12 4 10 2 14 16 17

277

260

257

257

254

(…)

5.1 Metoda pomiaru czasu

Pomiary czasu uzyskane zostały przy pomocy funkcji clock pochodzącej z biblioteki time.h.

Funkcja clock zwraca czas zużyty przez procesor w „clock ticks”. Następnie otrzymane tiki zostają przekonwertowane na milisekundy przy pomocy makra CLOCKS_PER_SEC. Za ostateczne obliczanie czasu odpowiedzialna jest następująca linia kodu:

```
cpu_time = ((double) (end - start)) / (CLOCKS_PER_SEC/1000);
```

5.2 Sprzęt

Procesor : Intel Xeon x5450, 4 rdzenie, 3.00 GHz

RAM: 8.00 GB

System: 64-bitowy system operacyjny, procesor x64

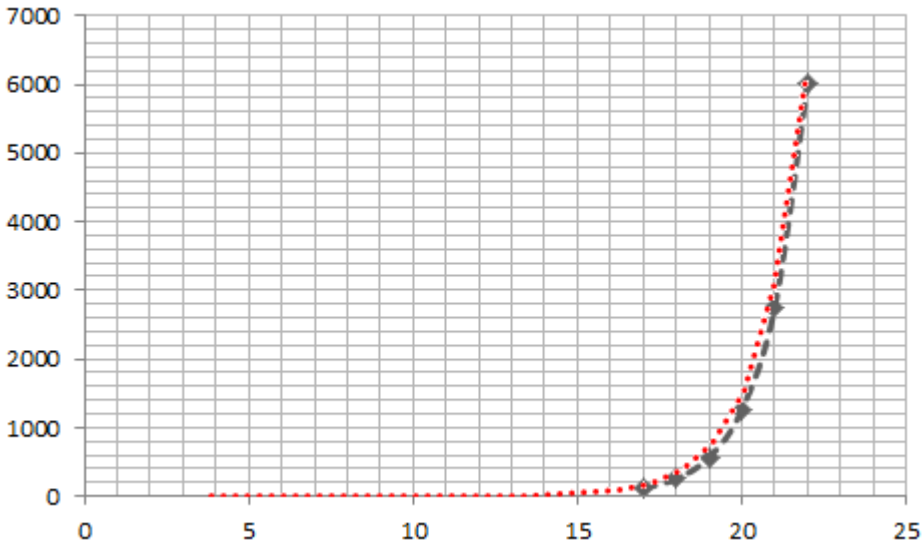
6. Wyniki

6.1 Pomiar czasu

Wyniki zwrócone w wyniku działania programu są przechowane w pliku final_test.csv.(jest to czysta wersja wyników zwróconych przez program). Natomiast ich wersja przedstawiona w przystępniejszej formie, wraz z wykresami została zawarta w pliku wyniki.xlsx.

Wyniki zostały przedstawione w postaci wykresu złożoności czasu [ms] uzyskania rozwiązania problemu od wielkości instancji [N] (rysunek 1.).

Na wykresie została umieszczona także krzywa funkcji 2^n zaznaczona kolorem czerwonym.

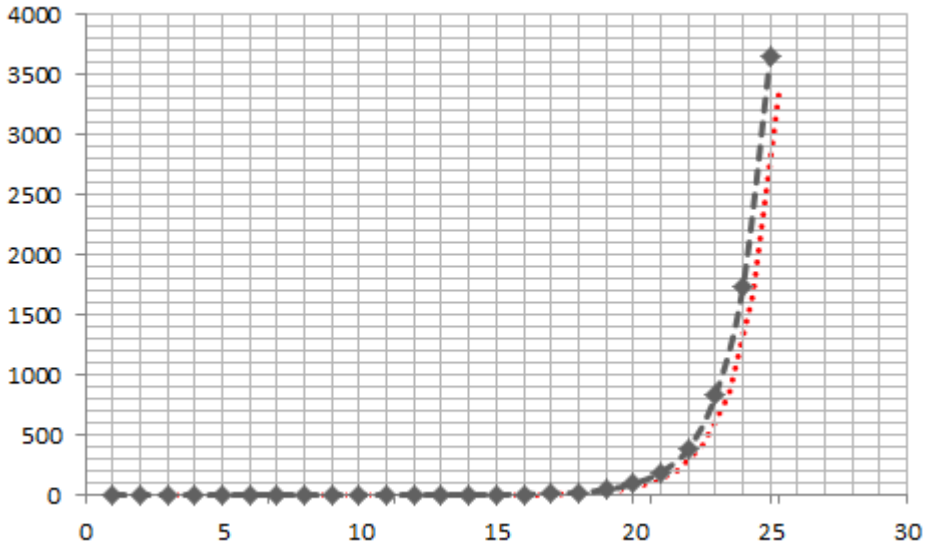


(rys 1.)

6. Pomiar zużycia pamięci

W celu zmierzenia zużycia pamięci generowane zostały przykładowe instancje zadania dla kolejnych wielkości problemu. Dla każdej takiej instancji zostało zmierzone zużycie pamięci przez proces. Nastepnie dane zostały zestawione w skoroszycie excel w postaci wykresu zużycia pamięci [MB] od rozmiaru danych N (rys. 2)

Na wykresie została umieszczona także krzywa funkcji 2^n zaznaczona kolorem czerwonym.



(rys. 2)

7. Wnioski

Zgodnie z oczekiwaniami krzywa wzrostu czasu względem wielkości instancji (rys. 1) jak i krzywa wzrostu zużycia pamięci względem wielkości instancji (rys. 2) przystaje do O(2^n). Potwierdza to nałożenie krzywej O(2^n), obie krzywe są bowiem zgodne co do kształtu.

Widzimy że algorytm ten jest znacznie wydajniejszy czasowo od metody brute force. Można wywnioskować to bezpośrednio z matematycznych obliczeń (wzór stirlinga) jak i z obserwacji wyników.

Niestety odbyło się to kosztem pamięci. Programowanie dynamiczne jest kosztowne pamięciowo. Mimo, iż czas obliczeń instancji 20-25 leży w przedziale kilku - kilkunastu sekund, to dla instancji 26+ następuje wyczerpanie pamięci komputera (8GB) i niemożliwe jest prowadzenie dalszych obliczeń.