Emilia Pawlaszek, 241279
GRUPA PN/17:05

Wrocław, 25 listopada 2019

Sprawozdanie z projektu przedmiotu „Projektowanie Efektywnych Algorytmów”

Rok akad. 2019/2020, kierunek: INF

Etap 1. Algorytmy dokładne.

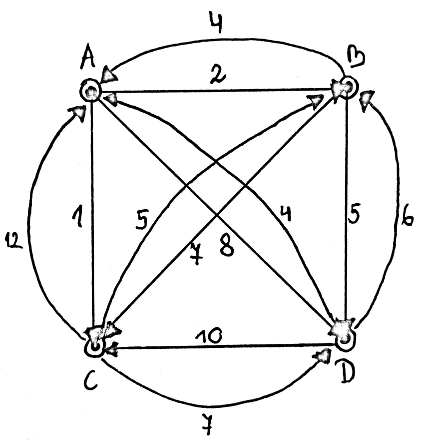
Spis treści

1 Asymetryczny problem komiwojażera (ATSP)

Komiwojażer musi odwiedzić pewną liczbę miast z danymi założeniami:

- W każdym mieście powinien znaleźć się tylko raz.
- Swoją wędrówkę rozpoczyna od wybranego miasta i tam też ją kończy.
- Wszystkie miasta są ze sobą połączone drogą.
- Każde przejście pomiędzy miastami ma pewien koszt, na przykład koszt podróży z miasta A do miasta B jest różny od kosztu podróży z miasta B do miasta A.
- Komiwojażer chce zminimalizować koszt całej swojej podróży między miastami.

Matematyczna wizualizacja problemu to asymetryczny graf zupełny, w którym każdy z wierzchołków ilustruje dane miasto. Odwiedzenie wszystkich miast odpowiada cyklowi, który przechodzi przez każdy wierzchołek danego grafu dokładnie raz. Cykl taki nazywamy cyklem Hamiltona. Poszukujemy więc w grafie cyklu Hamiltona o minimalnej sumie wag krawędzi.
Poniżej przedstawione są dwa obrazki, kolejno z przykładowym grafem posiadającym 4 wierzchołki i odpowiadającą mu macierzą z wagami.



(a) obraz 1

i \ j	A	B	C	D
A	-1	2	1	8
B	4	-1	7	5
C	12	5	-1	7
D	4	6	10	-1

(b) obraz 2

Rysunek 1: Asymetryczny graf zupełny z odpowiadającymi mu wagami.

Problem ten jest łatwy do rozwiązania dla małych instancji, jednak jego złożoność rośnie wraz z większą ilością wierzchołków. Problem ten jest NP-trudny.
Liczba cykli Hamiltona w grafie posiadającym n wierzchołków:
$L_H = (n-1) * (n-2) * ... * 3 * 2 * 1 = (n-1)!$
Wynik ten prowadzi do wykładniczej klasy złożoności obliczeniowej O(n!).

2 Opis algorytmów

2.1 Przegląd zupełny (Brute Force)

Metoda ta polega na rozważeniu wszystkich możliwych ścieżek w grafie i wybrania najlepszej z nich. Algorytm ten działa sprawnie dla małych instancji, dla grafu posiadającego maksymalnie 13 wierzchołków.

Kolejne etapy algorytmu:

1. Można przyjąć pierwszy wierzchołek jako wierzchołek początkowy (w grafie będzie szukany cykl więc i tak nie ma znaczenia jaki wierzchołek uznamy za początkowy).
2. Wygenerowanie (n-1)! permutacji wierzchołków.
3. Policzenie kosztu każdej permutacji oraz zachowanie informacji na temat najmniejszego dotychczasowego kosztu permutacji.
4. Wynikiem będzie permutacja o minimalnym koszcie.

2.2 Programowanie dynamiczne (Dynamic Programming)

Algorytm Helda-Karpa służy do rozwiązania problemu komiwojażera. Algorytm ten opiera się na programowaniu dynamicznym.

Z programowania dynamicznego możemy skorzystać wtedy, gdy jesteśmy w stanie podzielić nasz problem na mniejsze podproblemy. Wyniki rozwiązań podproblemów są zapamiętywane, więc nigdy ten sam podproblem nie jest liczony dwa razy, po prostu zwracany jest zapamiętany wynik. W metodzie tej korzysta się z rekurencyjnego wywołania funkcji.

Algorytm Helda-Karpa ma złożoność czasową O($n^2 2^n$) i złożoność pamięciową O($n2^n$). Jest to złożoność gorsza od wielomianowej, ale algorytm ten jest znacznie lepszy od algorytmu sprawdzającego wszystkie warianty (Brute Force).

Kolejne etapy algorytmu:

1. Stworzenie pomocniczego kontenera na zapamiętywane wyniki o wielkości [2 do N][N].
2. Sprawdzamy czy już wywołaliśmy algorytm z danymi parametrami, jeśli tak to korzystamy z naszego pomocniczego kontenera i zwracamy zapamiętaną na odpowiedniej pozycji wartość.
3. Tworzymy pamiętaną zmienną na minimalny wynik.
4. Sprawdzamy czy miasto nie jest odwiedzone, jeśli nie jest to dodajemy do naszej tymczasowej zmiennej obecną wagę i wynik wywołania rekurencyjnego algorytmu z kolejnymi parametrami.
5. Następnie wybieramy minimum z poprzedniego wyniku i tymczasowej zmiennej z obecnym wynikiem i zapisujemy do pamiętanej zmiennej.
6. Zwracamy najmniejszy wynik.

3 Implementacja algorytmów

3.1 Brute Force

Listing 1: Algorytm Brute Force

```
int findPath() {
    //wirzcholek poczatkowy
    int startVertex = 0;
    vector<int> vertex;
    //bez zerowego
    for (int i = 0; i < size; i++) {
        if (i != startVertex) {
            vertex.push_back(i);
        }
    }
    int minPath = INT_MAX;
    do {
        int currentPathWeigh = 0;
        int k = startVertex;
        for (int i = 0; i < vertex.size(); i++) {
            currentPathWeigh += graph[k][vertex[i]];
            k = vertex[i];
        }
         //dodajemy wage ostatniej krawadzi
        currentPathWeigh += graph[k][startVertex];
        minPath = min(minPath, currentPathWeigh);   //zamien
    }
     //zwraca true jesli moze zmienic obiekt w wieksza permutacje
    while (next_permutation(vertex.begin(), vertex.end()));
    return minPath;
}
```

3.2 Dynamic Programming

Listing 2: Algorytm Brute Force

```

for (int i = 0; i < (1 << size); i++) {
    for (int j = 0; j < size; j++) {
        dp[i][j] = -1;
    }
}
algorithm(1, 0);

int algorithm(int mask, int pos) {

    //jesli wszystkie miasta sa odwiedzone
    int visitedAll = (1 << size) - 1;
    if (mask == visitedAll) {
        return graph[pos][0]; //waga ostatniego polaczenia
    }
    //jezeli wywolalismy juz algorytm z tymi parametrami
    //to wynik bedzie zapisany na tej pozycji w tablicy dp
    if (dp[mask][pos] != -1) {
        return dp[mask][pos];
    }
    int ans = INT_MAX;
    for (int i = 0; i < size; i++) {
        if ((mask & (1 << i)) == 0){ //jesli nie jest odwiedzone
        //obecna waga + kolejne wywolanie
            int newAns = graph[pos][i] + algorithm(mask | (1 << i), i);
            ans = min(ans, newAns);
        }
    }
    return dp[mask][pos] = ans;
}
```

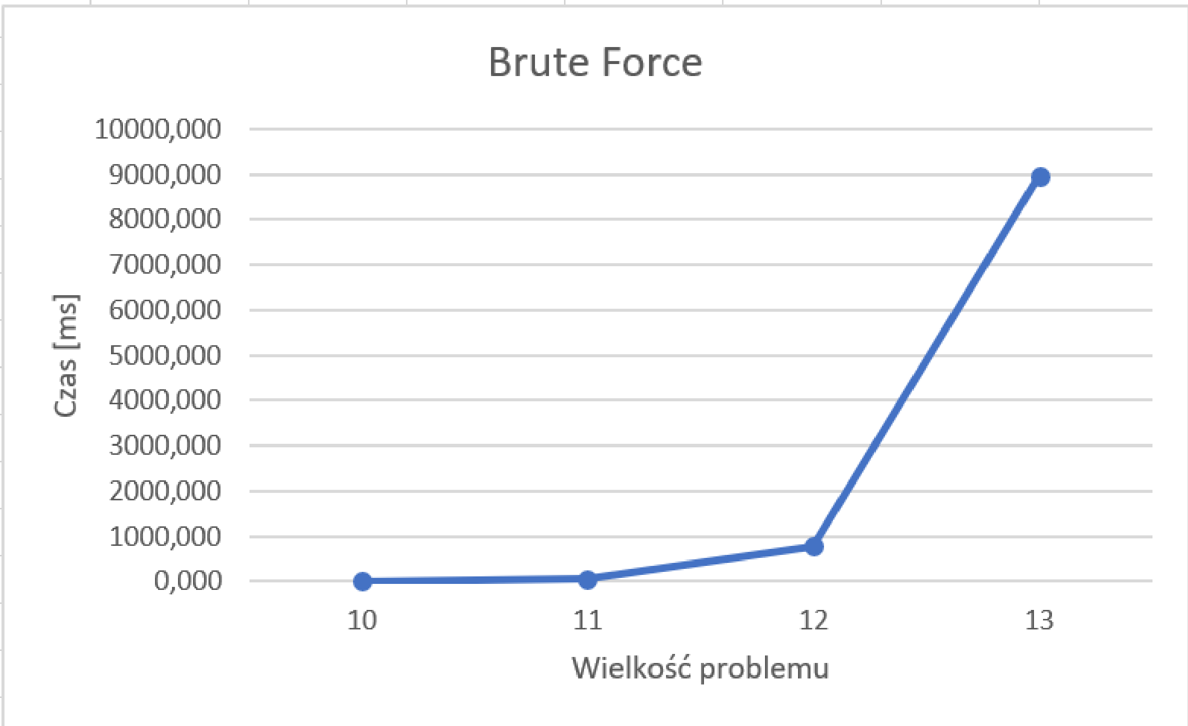
4 Badania

Obliczenia zastały wykonane na laptopie z procesorem i5-7200, kartą graficzną NVIDIA GeForce 940, 8GB RAM i DYSK SSD. Dla każdej wielkości problemu zostało wykonane 10 pomiarów i została wyciągnięta średnia.

4.1 Brute Force

Tablica 1: TableName

lp	wielkość problemu	czas [ms]	waga optymalnej drogi
1	10	5,942	212
2	11	64,289	202
3	12	785,209	264
4	13	8968,710	269

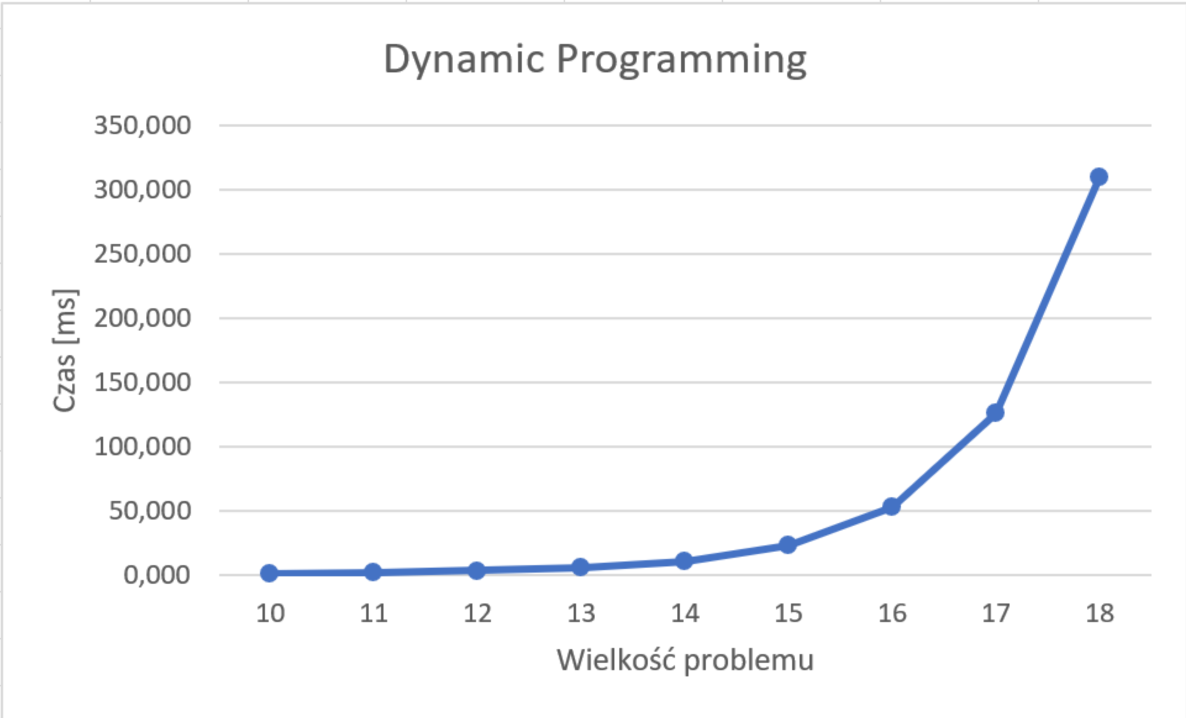


Rysunek 2: Wykres zależności czasu od wielkości problemu.

4.2 Dynamic Programming

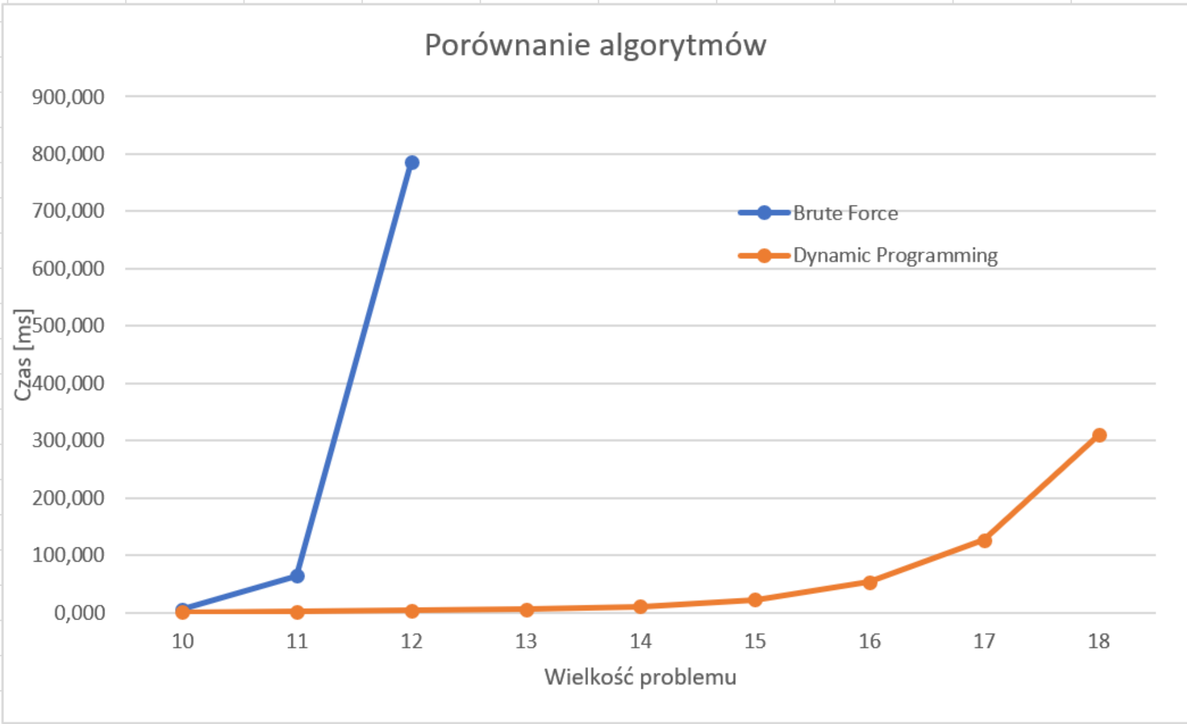
Tablica 2: TableName

lp	wielkość problemu	czas [ms]	waga optymalnej drogi
1	10	0,805	212
2	11	1,635	202
3	12	3,066	264
4	13	5,735	269
5	14	10,343	125
6	15	22,779	291
7	16	52,632	156
8	17	126,021	183
9	18	309,779	187



Rysunek 3: Wykres zależności czasu od wielkości problemu.

4.3 Porównanie algorytmów



Rysunek 4: Wykres zależności czasu od wielkości problemu.

5 Wnioski

Algorytm wykorzystujący programowanie dynamiczne okazał się dużo szybszy od przeglądu zupełnego. Podczas przeprowadzania badań należy mieć na uwadze aby w tle nie działały inne procesy. Należy też skorzystać z opcji 'Relase' podczas uruchamiania programu. Wyniki są wtedy dużo mniejsze.

Literatura

[1] Algorytmy i struktury danych: Problem komiwojażera.
`https://eduinf.waw.pl/inf/alg/001search/0140.php`

[2] Encyklopedia Algorytmów: Algorytm Helda-Karpa.
`http://algorytmy.ency.pl/artykul/algorytm_helda_karpa`