Projektowanie Efektywnych Algorytmów

Kierunek *Informatyka*	**Termin** *Czwartek 17:05*
Temat *Przegląd zupełny(Brute Force), programowanie dynamiczne.*	**Problem** *TSP*
Skład grupy *229218 Michał Honc*	**Nr grupy** *1*
Prowadzący *Mgr inż. Radosław Idzikowski*	**data** *13 listopada 2018*

1 Opis problemu

Problem Komiwojażera (Travelling Salesman Problem), W problemie mamy n miast oraz macierz nxn przejść pomiędzy miastami. Za zadanie mamy znaleźć jak najkrótszą ścieżkę przechodzącą przez wszystkie miasta wliczając w to powrót do miasta początkowego.

2 Metoda rozwiązania

2.1 Brute Force

Rozwiązanie problemu komiwojażera możemy uzyskać prostym algorytmem, który wyznacza wszystkie cykle Hamiltona, liczy sumę wag krawędzi i zwraca cykl o najmniejszej sumie wag. Rozwiązanie to jest o tyle dobre, iż zawsze zwróci cykl optymalny, a nie przybliżony. Również jest łatwe do zrozumienia i implementacji. Podstawowa wada to złożoność obliczeniowa O(n!), która umożliwia rozwiązanie problemu komiwojażera do około 20 wierzchołków.

Listing 1: Brute Force

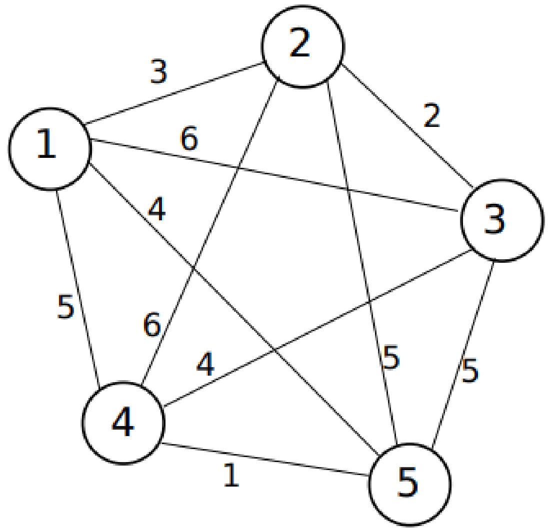
```
void BF::Brute_force(int v)
{
    int u;
    Ha_temp[qtemp++] = v;  // zapamietujemy biezacy wierzcholek na liccie pomocniczej

    if (qtemp < n)                    // jesli brak sciezki Hamiltona, to jej szukamy
    {
        visited[v] = true;            // Oznaczamy biezacy wierzcholek jako odwiedzony
        for (u = 0; u < n; u++)       // Przegladamy sasiadow wierzcholka v
            if (A[v][u] && !visited[u]) // Szukamy nieodwiedzonego jeszcze sasiada
            {
                dh += W[v][u];        // Dodajemy wage krawedzi v-u do sumy
                Brute_force(u);  // Rekurencyjnie wywolujemy szukanie cyklu Hamiltona
                dh -= W[v][u];        // Usuwamy wage krawedzi z sumy
            }
        visited[v] = false;           // Zwalniamy biezacy wierzcholek
    }
    else if (A[v0][v])                // Jesli znaleziona sciezka jest cyklem Hamiltona
    {
        dh += W[v][v0];               // to sprawdzamy, czy ma najmniejsza sume wag
        if (dh < d)
        {
            d = dh;                   // zapamietujemy te sume
            for (int u = 0; u < qtemp; u++)// oraz kopiujemy liste qtemp do q
                Ha[u] = Ha_temp[u];
            q = qtemp;
        }
        dh -= W[v][v0];               // Usuwamy wage krawedzi v-v0 z sumy
    }
    qtemp--;                          // Usuwamy biez cy wierzcholek ze scie ki
}
```

2.2 Programowanie dynamiczne

Programowanie dynamiczne (ang. dynamic programming) jest metodą rozwiązywania złożonych problemów, poprzez rozbicie ich na zbiór podproblemów o mniejszej złożoności, przy założeniu, że każdy podproblem rozważany jest jedynie raz, a wynik jego analizy przechowywany jest do wykorzystania w późniejszych obliczeniach. Posiada złożoność czasową O(n2 * 2n).

Przykład (start w wierzchołku 1)

S \ x	1	2	3	4	5
ϕ		3	6	5	4
{2}			5	9	8
{3}		8		10	11
{4}		11	9		6
{5}		9	9	5	
{2,3}				9	10
{2,4}			13		10
{2,5}			11	9	
{3,4}		11			11
{3,5}		11		12	
{4,5}		11	9		
{2,3,4}					10
{2,3,5}				11	
{2,4,5}			13		
{3,4,5}		11			
{2,3,4,5}	**14**				



- Przykładowo: $P(5, \{2,3,4\})=\min(\boldsymbol{P(4,\{2,3\})+w_{45}}, P(3,\{2,4\})+w_{35}, P(2, \{3,4\})+w_{25})$
- Trasa optymalna 1→ 2→ 3→ 4→ 5→ 1 ma długość 14

Rys.1 Przykład działania algorytmu programowania dynamicznego.

Listing 2: Programowanie dynamiczne

```
    for (int i = 0; i < n; ++i)
        dp[0][i] = dp[1 << 0][i] = INFTY;
    dp[1 << 0][0] = 0;
    for (int mask = 2; mask < (1 << n); ++mask)
        for (int i = 0; i < n; ++i) //podzial na podzbiory
        {
            dp[mask][i] = INFTY;
            if (mask & (1 << i))
                // AND maski i przesuniecia bitowego 1 o i bitow w prawo,
                // sprawdzenie czy bit znajdujacy sie na
                //odpowiedniej pozycji jest zapalony, czy nie
            {
                int mask1 = mask ^ (1 << i);
                //XOR maski i przesuniecia bitowego, zapala i-ty bit w masce
                for (int j = 0; j < n; ++j)
                    //obliczenie wartosci dla wszystkich podzbiorow
                    //znajdujacych sie w rozpatrywanym podzbiorze
                    if (mask1 & (1 << j))
                        //AND, sprawdzenie czy bit znajdujacy sie
                        //na odpowiedniej pozycji jest zapalony, czy nie
                        dp[mask][i] = min(dp[mask][i], dp[mask1][j] + t[j][i]);
                        //minimum danej sciezki
            }
        }
    int wynik = INFTY;
    for (int i = 0; i < n; ++i)
        wynik = min(wynik, dp[(1 << n) - 1][i] + t[i][0]);
    //rozpatrzenie przejscia przez ostatnia krawedz i
    //wybranie najmniejszej mozliwej wartosci

    cout << wynik << endl;
}
```

3 Eksperymenty obliczeniowe

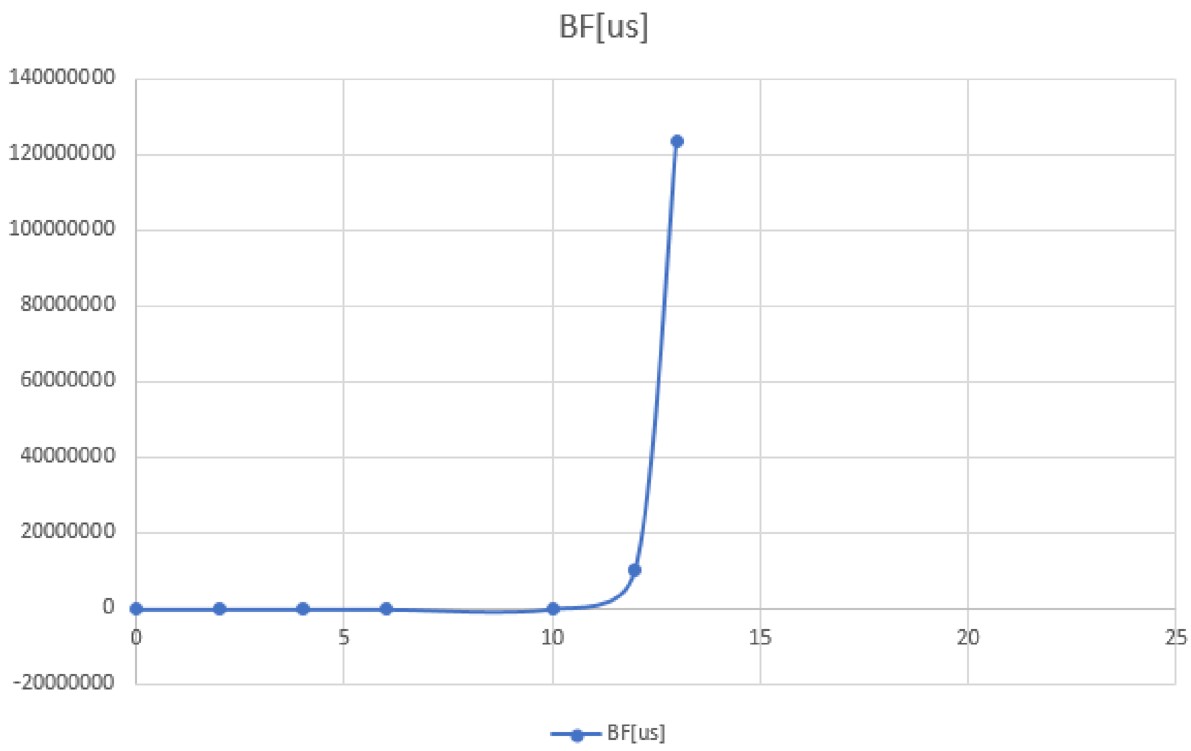
Obliczenia zastały wykonane na komputerze klasy PC z procesorem Intel Pentium, kartą graficzną zintegrowaną, 8GB RAM i DYSK SSD. Algorytmy operują na zmiennych integer 32bit. Pomiar czasu wykonany jest w mikrosekundach.

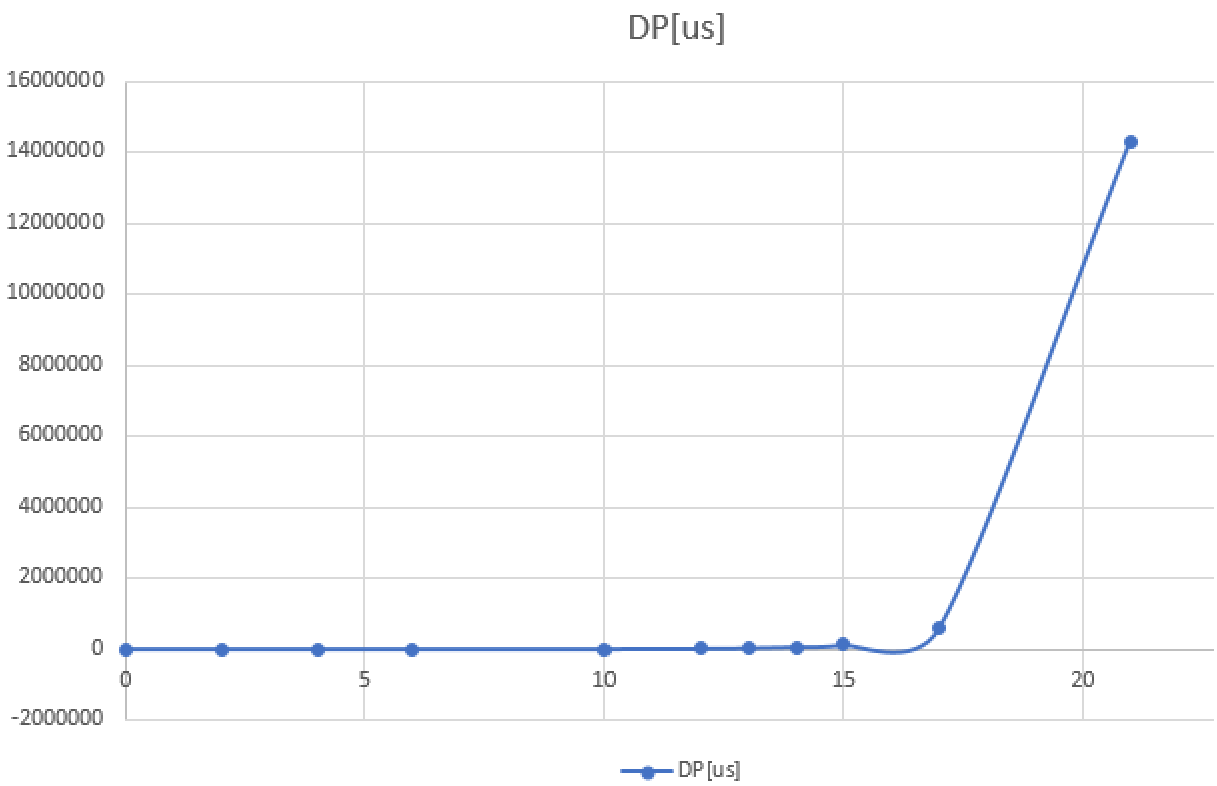
Wszystkie wyniki zebrano i przedstawiono w tabeli nr 1 gdzie:

- n - liczba wierzchołków,
- $BF[us]$ - czas wykonania Brute Force w mikrosekundach,
- $DP[us]$ - czas wykonania Programowania Dynamicznego w mikrosekundach

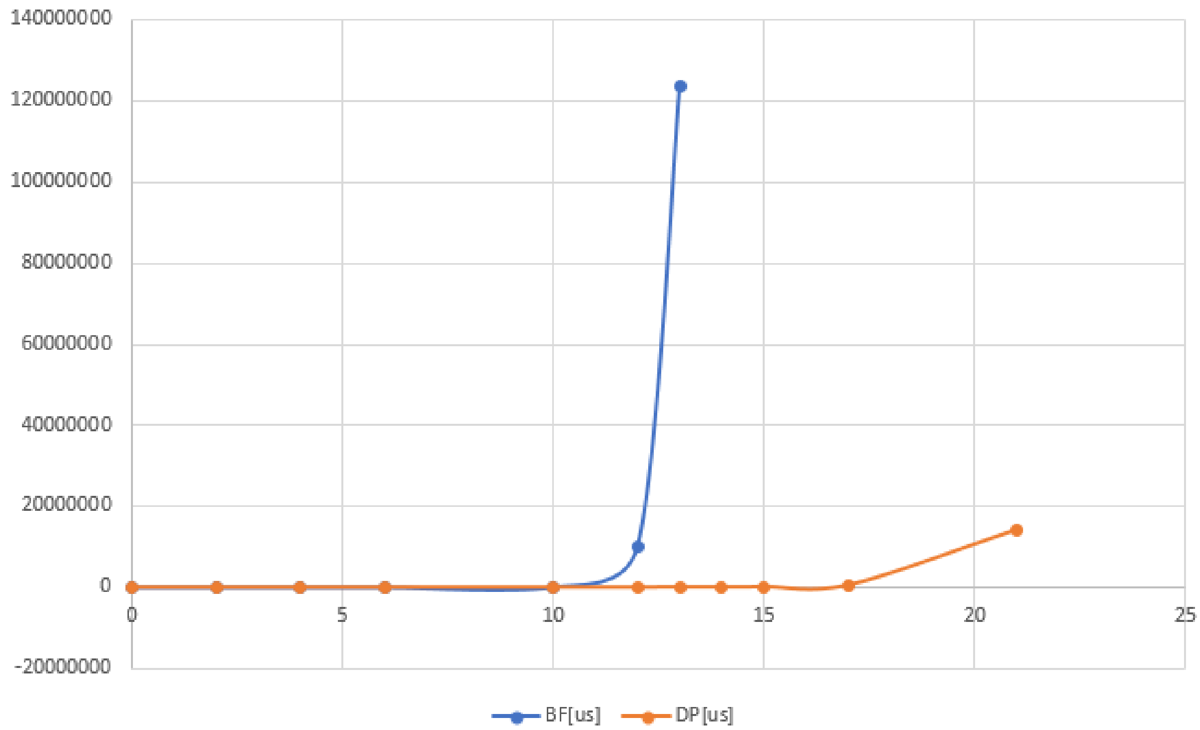
Tablica 1: Czas obliczeń TSP dla ustalonej liczby wierzchołków.

n	$BF[us]$	DP[us]
0	0	0
2	577	124
4	623	134
6	886	212
10	88386	1802
12	10197712	10383
13	123566609	23398
14		52670
15		116352
17		574111
21		14324367




DP[us]
16000000
14000000
12000000
10000000
8000000
6000000
4000000
2000000
0
-2000000
0
5
10
15
20
DP[us]

.png


140000000
120000000
100000000
80000000
60000000
40000000
20000000
0
-20000000
0
5
10
15
20
25
BF[us]
DP[us]

.png

4 Wnioski

Dla dużych zbiorów danych, algorytm przeglądu zupełnego jest całkowicie niewydajny, ze względu na jego złożoność obliczeniową. Algorytm wykorzystujący metodę programowania dynamicznego wykazuje się znacznie szybszym czasem wykonania, rosnącym znacząco wolniej wraz ze wzrostem zbioru danych.