Piotr Wodzyński
225968

Projektowanie efektywnych algorytmów.
Projekt 1
Programowanie dynamiczne dla problemu komiwojażera
Algorytm Held-Karpa

Dr inż. Jarosław Rudy

1. Wstęp teoretyczny

1.1 Problem komiwojażera - problem znalezienia minimalnego cyklu Hamiltona w grafie. Cyklem Hamiltona nazywamy taki cykl w grafie, w którym każdy wierzchołek grafu jest odwiedzony dokładnie raz. Dobrą wizualizacją tego problemu jest przykład podróżnika, który chce odwiedzić określoną liczbę krajów i zapłacić minimalną cenę za wszystkie bilety razem wzięte. (warto zauważyć, że np. cena biletu z Warszawy do Moskwy prawie nigdy nie jest taka sama jak cena biletu z Moskwy do Warszawy - w takim przypadku mówimy o asymetrycznym problemie komiwojażera). Jeśli mówimy o rozwiązaniu problemu komiwojażera podstawowym utrudnieniem jest spora złożoność obliczeniowa - np. dla metody przeglądu zupełnego wynosi ona O(n!)

1.2 Programowanie dynamiczne - metoda ta jest pewnym rozszerzeniem metody „dziel i zwyciężaj”. Polega na znalezieniu powtarzających się podproblemów, a następnie zapisaniu ich wyników obliczeń do tablicy pomocniczej. Wtórne wykorzystanie obliczonych już wyników prowadzi do widocznego obniżenia złożoności obliczeniowej, co prowadzi do szybszego rozwiązania danego problemu. Wadą takiego rozwiązania jest zwiększenie złożoności pamięciowej (wyniki obliczeń muszą być gdzieś zapisywane) względem innych metod.

1.3 Algorytm Held-Harpa - jeden z algorytmów programowania dynamicznego, który rozwiązuje problem komiwojażera. Jego złożoność pamięciowa to O($2^n n$), a obliczeniowa to O($2^n n^2$).

2. Opis algorytmu Held-Karpa na przykładzie

2.0 Wytłumaczenie oznaczeń funkcji

-*g(x, S)* - zaczynając od wierzchołka pierwszego, ścieżka minimalnego kosztu do wierzchołka x poprzez wierzchołki w zbiorze S

-c_{xy} - koszt przejścia z wierzchołka y do wierzchołka x

-p(x, S) - przedostatni wierzchołek przed dojściem do wierzchołka x. Jest to wierzchołek ze zbioru S.

2.0.1 Macierz reprezentująca połączenia w grafie przykładowym:

$$C = \begin{pmatrix} 0 & 2 & 9 & 10 \\ 1 & 0 & 6 & 4 \\ 15 & 7 & 0 & 8 \\ 6 & 3 & 12 & 0 \end{pmatrix}$$

2.1 Obliczenie ścieżki minimalnego kosztu (*g(x, S)*) dla pustego zbioru S

$g(2, \varnothing) = c_{21} = 1$
$g(3, \varnothing) = c_{31} = 15$
$g(4, \varnothing) = c_{41} = 6$

2.2 Obliczenie ścieżki minimalnego kosztu (g(x, S)) dla jednoelementowego zbioru S

- S = {2}
  $g(3,\{2\}) = c_{32} + g(2, \varnothing) = c_{32} + c_{21} = 7 + 1 = 8$ $p(3,\{2\}) = 2$
  $g(4,\{2\}) = c_{42} + g(2, \varnothing) = c_{42} + c_{21} = 3 + 1 = 4$ $p(4,\{2\}) = 2$
- S = {3}
  $g(2,\{3\}) = c_{23} + g(3, \varnothing) = c_{23} + c_{31} = 6 + 15 = 21$ $p(2,\{3\}) = 3$
  $g(4,\{3\}) = c_{43} + g(3, \varnothing) = c_{43} + c_{31} = 12 + 15 = 27$ $p(4,\{3\}) = 3$
- S = {4}
  $g(2,\{4\}) = c_{24} + g(4, \varnothing) = c_{24} + c_{41} = 4 + 6 = 10$ $p(2,\{4\}) = 4$
  $g(3,\{4\}) = c_{34} + g(4, \varnothing) = c_{34} + c_{41} = 8 + 6 = 14$ $p(3,\{4\}) = 4$

2.3 Obliczenie ścieżki minimalnego kosztu (g(x, S)) dla dwuelementowego zbioru S

- S={2,3}
  $g(4,\{2,3\}) = \min \{c_{42} + g(2,\{3\}), c_{43} + g(3,\{2\})\} = \min \{3+21, 12+8\} = \min \{24, 20\} = 20$
  $p(4,\{2,3\}) = 3$
- S={2,4}
  $g(3,\{2,4\}) = \min \{c_{32} + g(2,\{4\}), c_{34} + g(4,\{2\})\} = \min \{7+10, 8+4\} = \min \{17, 12\} = 12$
  $p(3,\{2,4\}) = 4$
- S={3,4}
  $g(2,\{3,4\}) = \min \{c_{23} + g(3,\{4\}), c_{24} + g(4,\{3\})\} = \min \{6+14, 4+27\} = \min \{20, 31\} = 20$
  $p(2,\{3,4\}) = 3$

2.4 Obliczenie ścieżki minimalnego kosztu (g(x,S)) dla trójelementowego zbioru S = {2,3,4}

$f = g(1,\{2,3,4\}) = \min \{ c_{12} + g(2,\{3,4\}), c_{13} + g(3,\{2,4\}), c_{14} + g(4,\{2,3\}) \}$
$= \min \{2 + 20, 9 + 12, 10 + 20\} = \min \{22, 21, 30\} = 21$

p(1,{2,3,4}) = 3

2.5 Zbiór S w punkcie 2.4 zawierał już wszystkie wierzchołki poza wierzchołkiem startowym dlatego na tym etapie można zakończyć działanie algorytmu. Kolejnym krokiem jest odtworzenie ścieżki grafu poprzez skorzystanie z funkcji p:

- p(1,{2,3,4} = 3
- p(3, {2,4}) = 4
- p(4, {2}) = 2

Z powyższych funkcji wynika że minimalnym cyklem Hamiltona w powyższym grafie jest cykl 1 → 3 → 4 → 2 → 1, a jego koszt to 21

3. Implementacja algorytmu w języku Java

```
public class Held_KarpAlgorithm {

        private ArrayList<Integer> outputArray = new ArrayList<Integer>();
        /*stała wielkość tablicy ze względu na lepszą wydajność (w stosunku do arraylist) przy
założeniu że nie będą dodawane i usuwane elementy*/
        private int npow, N, distances[][], connections[][], path[][];
        public static long time;

        public Held_KarpAlgorithm() {
        }

        public ArrayList<Integer> computeTSP(int[][] graphMatrix, int n) {
            long start = System.nanoTime();

            N = n;
            npow = (int) Math.pow(2, n);
            connections = new int[n][npow];
            path = new int[n][npow];
            distances = graphMatrix;

            int i, j;

            for (i = 0; i < n; i++) {
                for (j = 0; j < npow; j++) {
                    connections[i][j] = -1; //zapełnienie tablicy wartościami aby nie były losowe
                    path[i][j] = -1;
                }
            }
```

```
            for (i = 0; i < n; i++) {
                connections[i][0] = graphMatrix[i][0]; //zainicjonowanie połączeń z pustym zestawem.
Krok pierwszy w algorytmi Held-Karpa
            }
            int result = tsp(0, npow - 2);
            outputArray.add(0);
            getPath(0, npow - 2);
            outputArray.add(result);

            long end = System.nanoTime();
            time = (end - start) / 1000;
            outputArray.add((int)time/1000);
            return outputArray;
        }

        private int tsp(int start, int set) {
            int masked, mask, result = -1, temp;
          //  System.out.print("Start: "+start+", set: "+set+"\n");
            if (connections[start][set] != -1) {
                return connections[start][set]; //sprawdznie czy dane połączenie już istnieje w
pamięci - cecha harakterystyczna dla programowania dynamicznego
            } else {
                for (int x = 0; x < N; x++) {
                    mask = npow - 1 - (int) Math.pow(2, x);
                    /*W poniższej linijce mamy iloczyn bitowy aby sprawdzić wszystkie opcje ze
zbioru połączenia*/
                    masked = set & mask;
                    if (masked != set) {    //jeśli dana opcja istnieje w zbiorze połączeń wykonać
poniższy kod
                        temp = distances[start][x] + tsp(x, masked); //przypisanie do zmiennej
tymczasowej nowo wygenerowanego połączenia
                      //  System.out.print("Temp: "+temp+"\n");
                        if (result == -1 || result > temp) {
                            result = temp; //przypianie do wyniku najmniejszej zmiennej tymczasowej
- odpowiada to połączeniu o najmniejszej wadze
                            path[start][set] = x;   //zapisanie następnego miasta dla tego zbioru
połączeń
                        }
                    }
```

```
                }
                connections[start][set] = result;   //zapisanie najbardziej wydajnego połączenia dla
danego miasta z danym zbiorem
                return result;
            }
        }

        private void getPath(int start, int set) {
            if (path[start][set] == -1) {
                return;
            }
            //odtworzenie ścieżki połączeń przez sprawdzenie kolejnych połączeń ze zbioru path -
także za pomocą iloczynów bitowych

            int x = path[start][set];
            int mask = npow - 1 - (int) Math.pow(2, x);
            int masked = set & mask;

            outputArray.add(x);
            getPath(x, masked);
        }
}
```

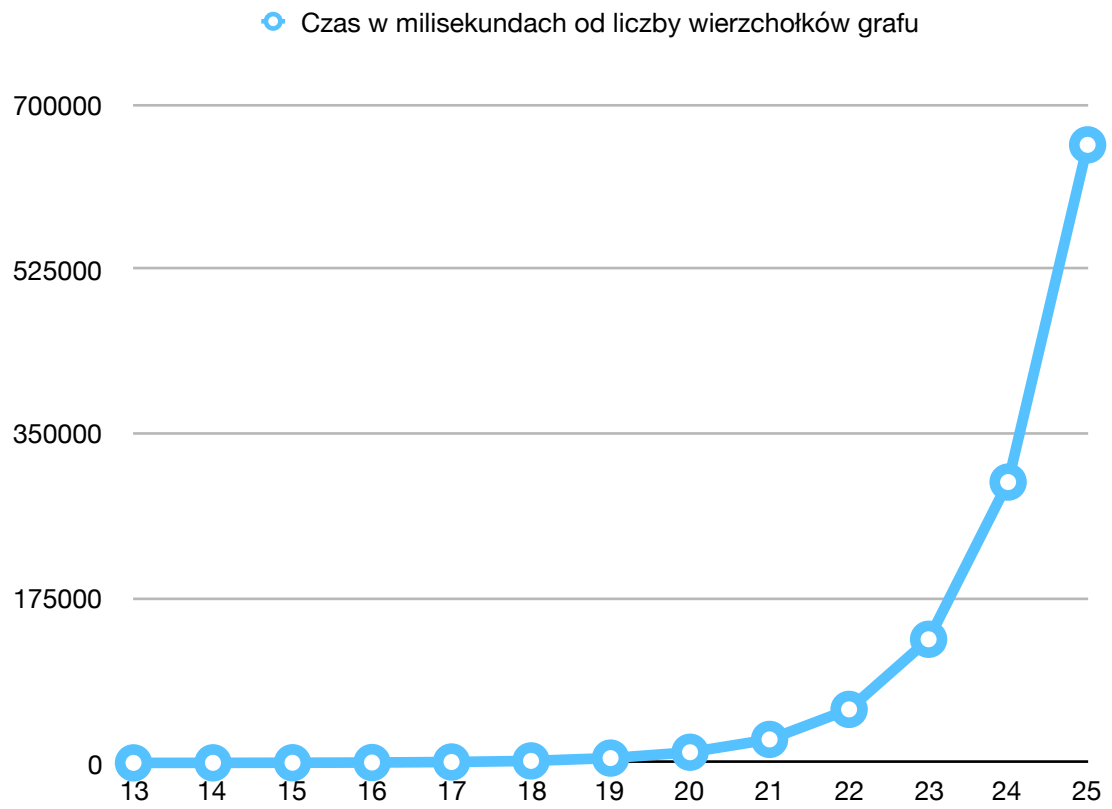
4. Pomiary czasów dla instancji ze strony

Jedynie cztery poniższe instancje były możliwe do wykonania ze względu na złożoność obliczeniową i pamięciową. Obliczone koszty minimalne zgadzały się z minimalnymi kosztami, które można znaleźć na stronie, z której pobrano instancje.

Nazwa pliku	Liczba wierzchołków	Koszt minimalny	Czas podany w milisekundach
gr17.tsp	17	2085	963
gr21.tsp	21	2707	26242
gr24.tsp	24	1272	296399
br17.atsp	17	39	975

5. Pomiary czasów dla instancji asymetrycznych wygenerowanych przeze mnie

Ze względu na dużą złożoność pamięciową i czasową dokonałem okrojenia grafu z pliku ftv33.atsp. Okrojenie polegało na wybraniu np. Tylko określonej liczby n wierzchołków z grafu i rozwiązaniu problemu komiwojażera dla tej liczby wierzchołków.



Liczba wierzchołków	Czas podany w milisekundach	Wartość oczekiwana	Błąd względny
13	31	43	0,2852073107
14	74	100	0,2643846797
15	170	230	0,2639428687
16	400	525	0,2389115141
17	964	1186	0,1876104632
18	2143	2660	0,1945616665
19	5112	5929	0,1377984842
20	11341	13139	0,1368495481
21	24969	28971	0,1381578955
22	56894	63593	0,1053436298

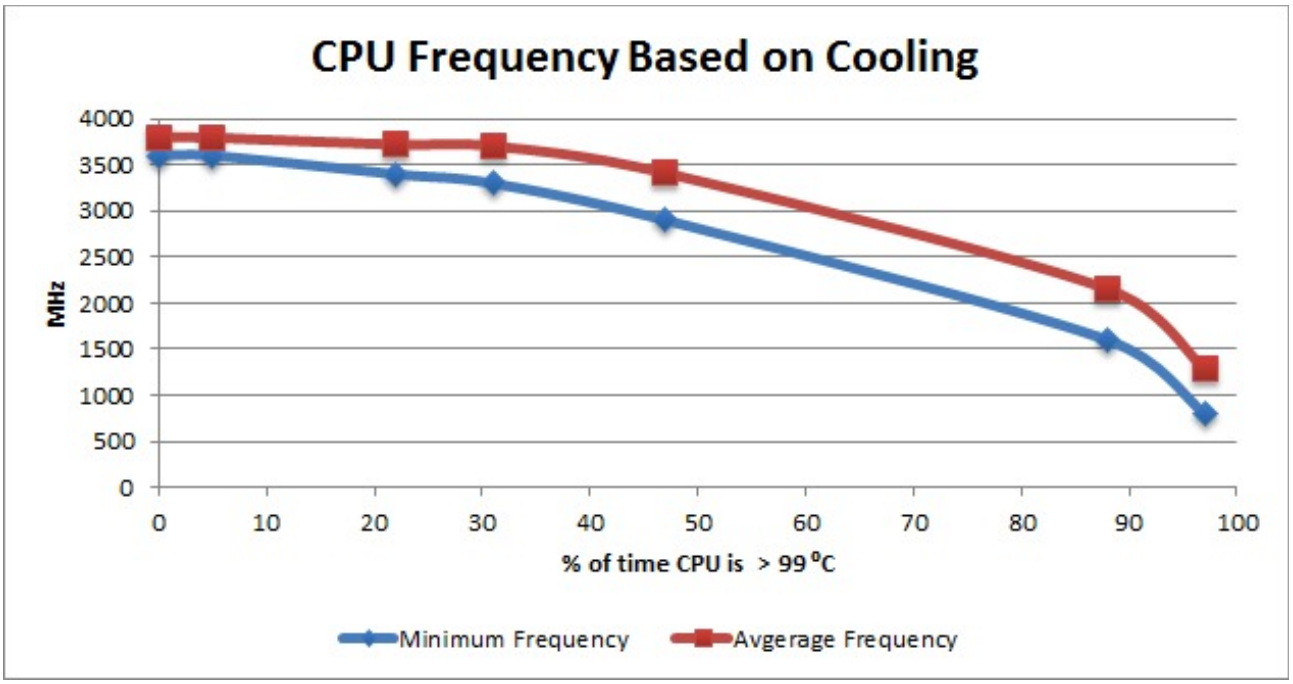
Liczba wierzchołków	Czas podany w milisekundach	Wartość oczekiwana	Błąd względny
23	131407	139011	0,0547036421
24	298512	302724	0,0139147304
25	656956	656953	0

Sposób obliczenia wartości oczekiwanej:
Korzystając z złożoności obliczeniowej stworzyłem poniższy wzór:

$$n^2 * 2^n * x = t$$

t - czas
x - stała czasowa pojedynczego obliczenia
n - liczba wierzchołków
Kolejnym krokiem było wyliczenie stałej czasowej pojedynczego obliczenia. Do tego użyłem czasów operacji, która rozwiązywała problem komiwojażera dla największej liczby instancji. Następnie obliczyłem przewidywane czasy dla reszty operacji.

6. Wnioski
Powyższy kod spełniał swoje zadanie w czasie charakterystycznym dla złożoności obliczeniowej tego algorytmu. Jedną z przyczyn rozbieżności poza charakterystyką tego algorytmu było najprawdopodobniej działanie Garbage Collectora w tyle, który jest charakterystyczny dla języka Java. Dodatkowym czynnikiem może być zwolnienie taktowania procesora na co ma wpływ temperatura:



Jeśli chodzi natomiast o zużycie pamięci, to powyższe rozwiązanie niestety zapełnia także pamięć, która nie będzie w przyszłości używana, jednak tablica o stałym rozmiarze (a nie np. Array Lista) pozytywnie wpływa na wydajność obliczeniową algorytmu.

Bibliografia:

https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm
https://github.com/evandrix/SPOJ/blob/master/DP_Main112/Solving-Traveling-Salesman-Problem-by-Dynamic-Programming-Approach-in-Java.pdf
https://pl.wikipedia.org/wiki/Problem_komiwoja%C5%BCera
http://www.algorytm.org/kurs-algorytmiki/programowanie-dynamiczne.html
https://www.pugetsystems.com/labs/articles/Impact-of-Temperature-on-Intel-CPU-Performance-606/ - taktowanie, a temperatura procesora.