Wykonała: Marta Głowacka 234999

Prowadzący mgr. inż. Antoni Sterna

# Projektowanie efektywnych algorytmów

## Zadanie 1

## Problem komiwojażera

### Wstęp teoretyczny

### 1.1 Opis rozpatrywanego problemu

Komiwojażer zaczyna swoją podróż z miasta rodzimego. Chce odwiedzić wszystkie miasta i wrócić na końcu do miasta, z którego wyruszył. Problem komiwojażera dotyczy znalezienia najkrótszej trasy jego podróży.

Ujmując problem w świetle teorii grafów, problem komiwojażera (TSP – Travelling Salesman Problem) dotyczy grafu pełnego i szukania w nim cyklu hamiltonowskiego (droga, w której każdy wierzchołek grafu występuje tylko raz) o minimalnej sumie wag krawędzi.

Rozróżniamy dwie wersje problemu TSP: symetryczny (odległość z miasta A do B jest taka sama jak z B do A) oraz asymetryczny (odległości mogą być różne). W programie zaprogramowane algorytmy mogą rozwiązywać oba problemy.

### 1.2 Opis algorytmu

Za start podróży przyjmujemy miasto „zerowe” (wierzchołek o indeksie 0). Zakładamy, że graf ma więcej niż 2 wierzchołki (program obsługuje przypadek, jeśli podano inaczej).

### Brute Force:

Algorytm generujący (n-1)! permutacji (różnych ścieżek, czyli wariantów kolejnego odwiedzania miast). Zwraca permutację o koszcie minimalnym trasy.

Złożoność czasowa O(n!)

### Dynamic Programming:

Programowanie dynamiczne to przydatna technika do rozwiązywania niektórych problemów m.in. TSP. Liczymy częściowe rozwiązania, których wyniki zapisujemy, aby następnie użyć ich parokrotnie, gdy częściowe rozwiązanie jest potrzebne.

Liczymy najlepsze rozwiązanie dla każdej podścieżki o długości N, używając informacji już znanych na temat ścieżek o długości N-1.

1. Rozwiązanie problemu dla n = 2. Wierzchołek startowy to 0. Liczymy dla niego odległości (0-1, 0-2, itd.) do każdego z pozostałych wierzchołków, a następnie wybieramy najmniejszą wartość.
2. Rozwiązanie problemu dla n = 3. Aby znaleźć najkrótszą drogę długości trzy, musimy gromadzić z każdego rozwiązania dla n = 2 dwie rzeczy: zbiór odwiedzonych wierzchołków w podścieżce (2n możliwości) oraz indeks ostatnio odwiedzonego wierzchołka w ścieżce (N możliwości).
3. Zbiór odwiedzonych wierzchołków przechowujemy w 32-bitowej zmiennej typu integer (zamiast tablicy wartości boolean) np. uzyskujemy 0…0011 (=3), jeśli odwiedziliśmy wierzchołki 0 i 1. Pozwala to na późniejsze szybkie operacje logiczne. Tworzymy tablicę dwuwymiarową memory o rozmiarze N na 2n. W każdym wierszu odpowiadającym wierzchołkowi są 32 bity zmiennej typu integer.
4. Rozwiązanie problemu dla 3 <= n <= N. Bierzemy rozwiązania dla podścieżek dla n-1 i dodajemy kolejną krawędź od ostatnio odwiedzonego wierzchołka (z tego powodu trzymamy go w pamięci) do wierzchołka jeszcze nieodwiedzonego. Ten proces się powtarza dla coraz większych ścieżek. Kończy się, gdy wszystkie podścieżki będą długości N.
5. Od końca rekonstruujemy ścieżkę. Zaczynamy od stanu końcowego (end\_state złożonego z samych jedynek – oznacza to, że wszystkie wierzchołki odwiedzono) w tablicy memory dla każdego potencjalnie końcowego wierzchołka. Kończymy na wierzchołku startowym.

### Branch and Bound:

Algorytm ten, uaktualnia dolną granicę minimalnej trasy, co sprawia, że możemy część obliczeń pominąć – zmniejszamy w ten sposób czas wykonania się algorytmu. Ważnym pojęciem w B&B jest tzw. drzewo przeszukiwań (stanów). Jest to kontener na dane w postaci zakorzenionego drzewa.

Z algorytmem Branch and Bound wiążą się pewne pojęcia warte wyjaśnienia:

górna granica - wartość, której na pewno nie przekroczy funkcja celu dla żadnego z rozwiązań, w których ustalono pewne elementy

Źródło: http://www.cs.put.poznan.pl/mkomosinski/materialy/optymalizacja/BB\_DP.pdf

węzeł obiecujący - węzeł, którego granica jest lepsza niż wartość najlepszego znalezionego dotąd rozwiązania

Źródło: <https://www.ii.uni.wroc.pl/~prz/2011lato/ah/opracowania/met_podz_ogr.opr.pdf>

Istnieją dwa podejścia do wdrożenia algorytmu B&B poprzez zaimplementowanie odmiennej budowy drzewa: **pierwszy** - polega na dołączeniu do korzenia wszystkich pozostałych wierzchołków jako dzieci, a następnie, idąc w głąb, doczepiania do dzieci kolejnych wierzchołków niewykorzystanych jeszcze na drodze od rodzica. W wyniku tego wierzchołki drzewa mogą mieć różną liczbę dzieci, a trasy od rodzica do liścia odpowiadają permutacjom wierzchołków grafu; **drugi** - polega na rozwiązaniu problemu, które zawierają pewne łuki i nie zawierają innych (wybranych) łuków np. wierzchołkowi odpowiadają trasy z łukami (a,b) i (c,d), ale bez łuków (a,c), (c,e), (b,c). Każdy węzeł ma dwóch synów.

Zastosowałam pierwsze podejście budowy drzewa, które reprezentuje wszystkie możliwe ścieżki, jakie algorytm może obrać. Zaczynamy w korzeniu i przechodząc w głąb drzewa po kolejnych (obiecujących, nie wszystkich) węzłach, dochodzimy do liścia. Liczymy rozwiązanie i kontynuujemy liczenie dla najbardziej obiecującego węzła – określamy to na podstawie ograniczenia (granicy/ kosztu), jaki przypisujemy każdemu wierzchołkowi - jeśli dla danego wierzchołka drzewa górna granica jest niższa od znanego już rozwiązania, jego gałąź pomijamy.

Drzewo przechodzimy wszerz. Dla każdego wierzchołka budujemy macierz na podstawie rodzica, którą uaktualniamy i redukujemy. Na podstawie kosztu redukcji, kosztu rodzica oraz odległości między wierzchołkami (rodzic, dziecko) liczymy koszt wierzchołka drzewa.

### 1.3 Oszacowanie złożoności obliczeniowej

Problem TSP jest NP-trudny tzn. że nie istnieją dla niego algorytmy o wielomianowej złożoności obliczeniowej. Aby znaleźć rozwiązanie musimy użyć któregoś z algorytmów przybliżonych (takie, które w jak najkrótszym czasie znajdują rozwiązanie równe optymalnemu).

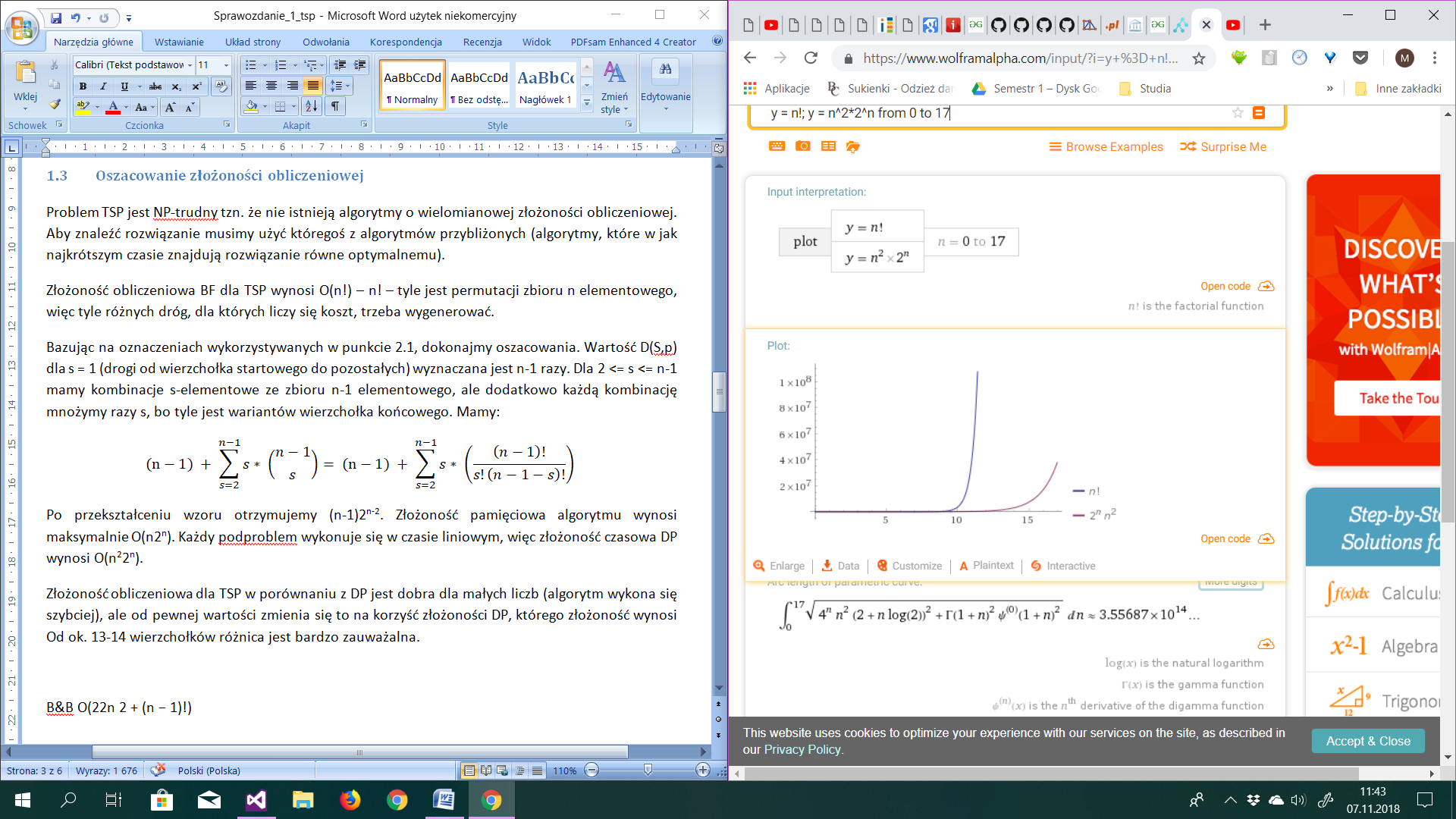
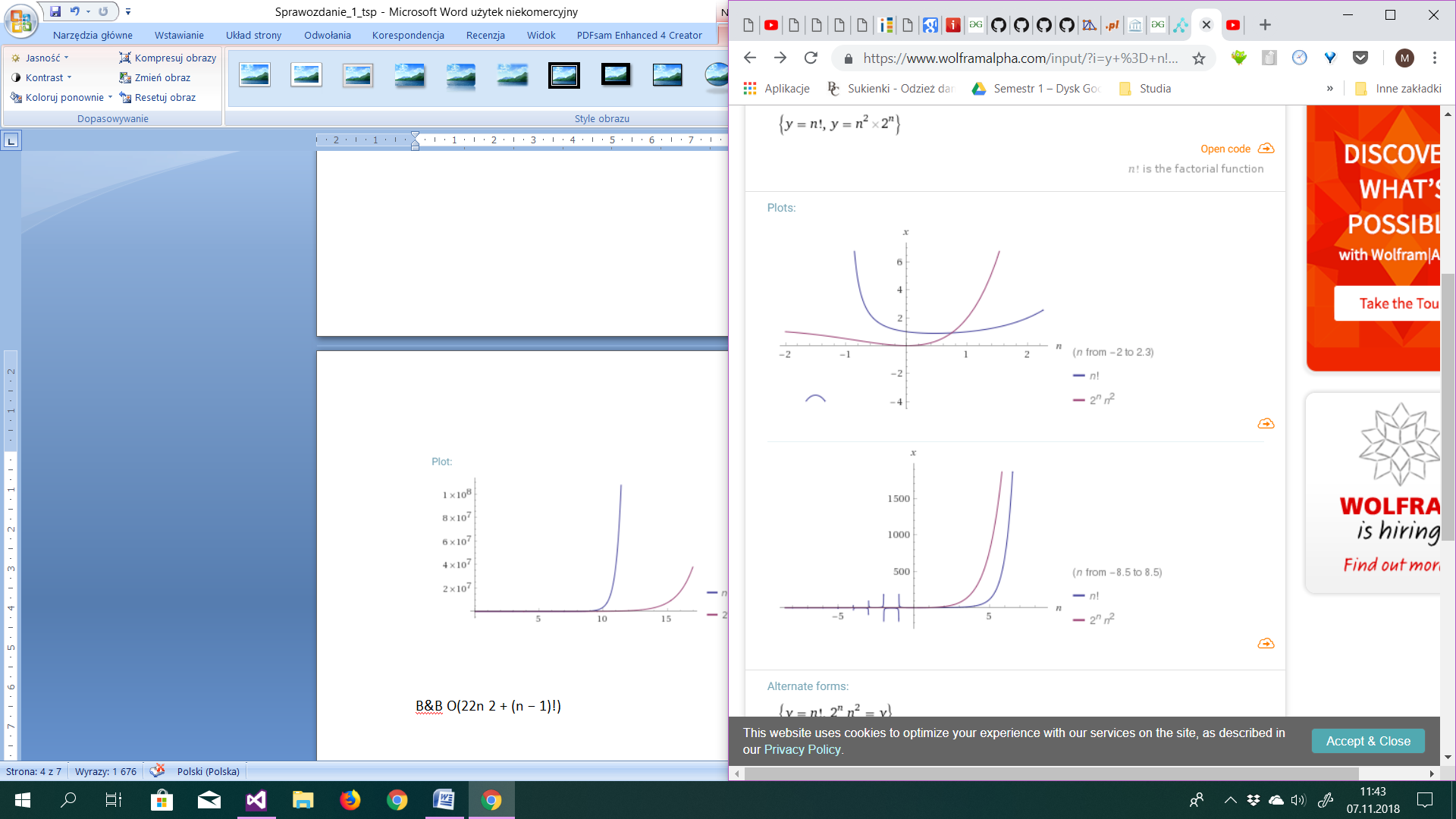
Złożoność obliczeniowa BF dla TSP wynosi O(n!) – n! – tyle jest permutacji zbioru n elementowego, więc tyle różnych dróg, dla których liczy się koszt, trzeba wygenerować.

Bazując na oznaczeniach wykorzystywanych w punkcie 2.1, dokonajmy oszacowania. Wartość D(S,p) dla s = 1 (drogi od wierzchołka startowego do pozostałych) wyznaczana jest n-1 razy. Dla 2 <= s <= n-1 mamy kombinacje s-elementowe ze zbioru n-1 elementowego, ale dodatkowo każdą kombinację mnożymy razy s, bo tyle jest wariantów wierzchołka końcowego. Mamy:

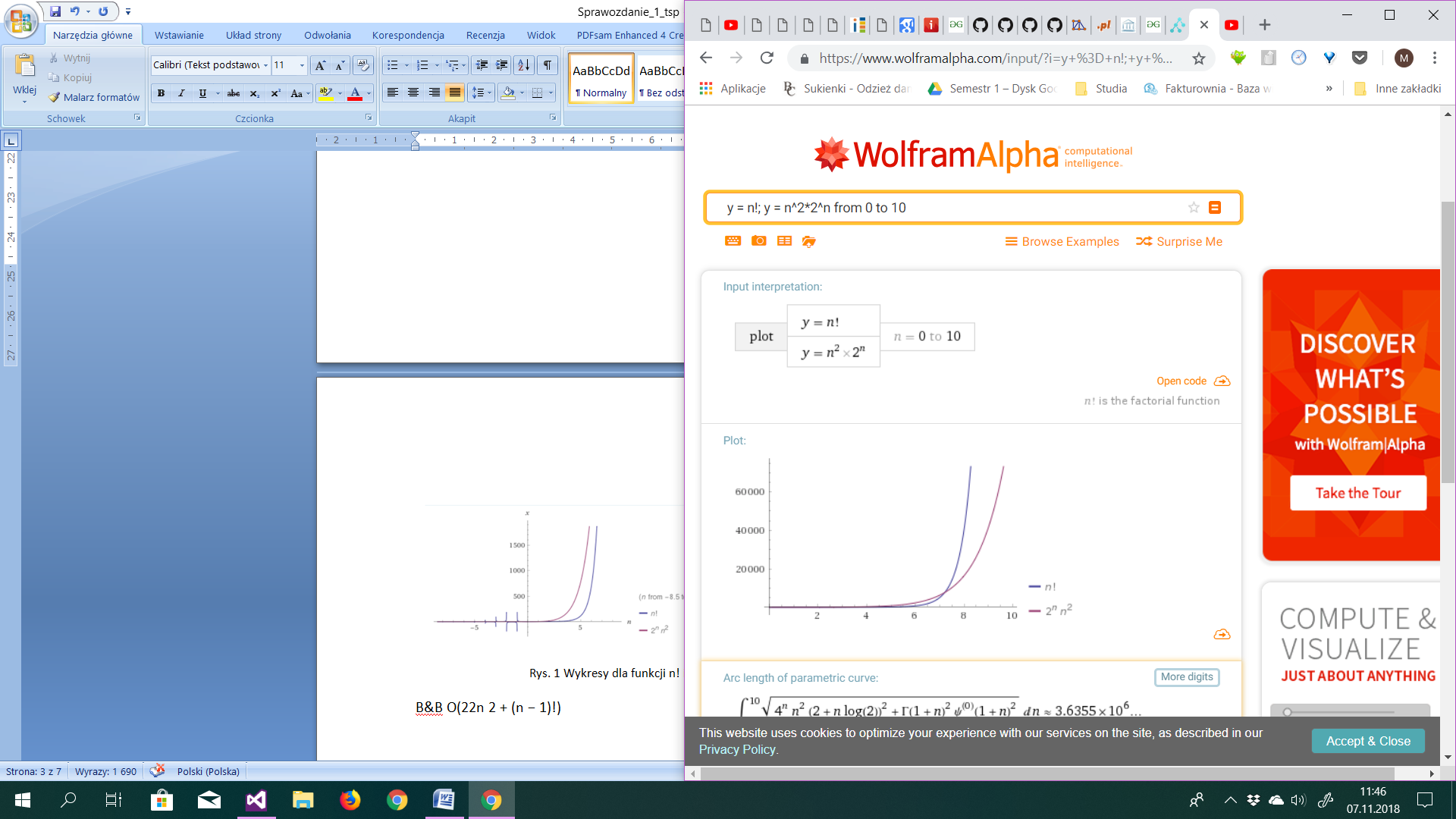
Po przekształceniu wzoru otrzymujemy (n-1)2n-2. Złożoność pamięciowa algorytmu wynosi maksymalnie O(n2n).

Każdy podproblem wykonuje się w czasie liniowym, więc złożoność czasowa DP wynosi O(n22n). Można to zrozumieć w ten sposób, że jest n możliwych wierzchołków startowych i 2n możliwych podgrafów – funkcja zostanie wywołana maksymalnie n\*2n raza. Każde wywołanie ma złożoność maksymalnie O(n), bo wierzchołek ma maksymalnie n sąsiadów. Stąd złożoność czasowa wynosi O(n22n).

Złożoność obliczeniowa dla TSP w porównaniu z DP jest dobra dla małych liczb (algorytm wykona się szybciej), ale od pewnej wartości zmienia się to na korzyść złożoności DP, którego złożoność wynosi Od ok. powyżej 8 wierzchołków różnica jest bardzo zauważalna.



Rys. 1 Wykresy dla funkcji n! i n22n dla przedziału argumentów (-7,7) i (0,17).



Rys.2 Miejsce przecięcia dwóch wykresów n! i n22n

Najtrudniej spotkać się z określeniem złożoności obliczeniowej dla algorytmu Branch and Bound. Z artykułu *Branch And Bound—Why Does It Work?* (link w bibliografii), w którym autorzy odnieśli się do złożoności czasowej wynika, że nie jest to problem trywialny, a raczej przypomina swoją złożonością grę w szachy. Wszystko zależy od funkcji, która dobierze granicę i na tej podstawie oszczędność czasu jest trudna do przewidzenia.

W artykule *Branch-and-Bound for the Travelling Salesman Problem* będącym w posiadaniu Ecole Polytechnique w Palaiseau we Francji znalazłam najbardziej konkretną odpowiedź w rozdziale Branch-and-Bound – A pure branching algorytm – Complexity, która wynosi . Zakładam, że wyliczona była dla jakiegoś szczególnego przypadku funkcji ograniczającej.

### Przykład praktyczny

4

6

6

1

3

11

5

4

1

2

9

-4

Dana jest macierz sąsiedztwa dla grafu o 4 wierzchołkach:

Kosz minimalny dla problemu TSP: 9

Trasa: A -> D -> C -> B -> A

#### 2.1 Brute Force:

Należy wypisać wszystkie permutacje i sprawdzić dla której koszt drogi jest najmniejszy.

|  |  |  |  |
| --- | --- | --- | --- |
| ABCD 18 | BACD 11 | CABD 15 | DABC 18 |
| ABDC 15 | **BADC 9** | CADB 24 | DACB 19 |
| ACBD 19 | BCAD 24 | **CBAD 9** | DBAC 11 |
| ACDB 11 | BCDA 18 | CBDA 19 | DBCA 24 |
| ADBC 24 | BDAC 19 | CDAB 18 | DCAB 15 |
| **ADCB 9** | BDCA 15 | CDBA 11 | **DCBA 9** |

Zakładając, że zaczynamy od wierzchołka A, wybieramy zestaw ADCB z pierwszej kolumny i doklejamy do niego na końcu wierzchołek startowy A.

#### 2.2 Dynamic programming:

Korzystając z algorytmu Bellmana-Helda-Karpa opartego na programowaniu dynamicznym, rozwiążemy ten sam problem.

Jako di,j oznaczmy odległość między wierzchołkami i oraz j.

Oznaczmy jako D(S, p) optymalną długość ścieżki wychodzącej z punktu 1 i przechodzącej przez wszystkie punkty zbioru S tak, aby zakończyć się w punkcie p (p musi należeć do S).

Wartość D(S, p) wyznaczamy następująco:

* Jeśli s=1, to D(S, p) = d1,p,
* Jeśli s>1, to D(S, p) = minx∈ (S-{p})( D(S−{p}, x) + dx,p)

Na końcu wyznacza się rozwiązanie całego problemu. W tym celu należy znaleźć poprzednika punktu 1 korzystając ze wzoru: minx∈{2, …, n}( D({2, …, n}, x) + dx,1)

Źródło: <http://algorytmy.ency.pl/artykul/algorytm_helda_karpa>

Odległości od wierzchołka startowego do pozostałych. Wyznaczamy wartości D(S, p) dla jednoelementowych zbiorów S (s=1)

* *D({2}, 2) = dA,B = 4*
* *D({3}, 3) = dA,C = 1*
* *D({4}, 4) = dA,4 = 9*

Wyznaczamy wartości D(S, p) dla dwuelementowych zbiorów S (s=2).

* *D({B, C}, B) = min( D({C}, C) + dC,B ) = min(1 + 1) = min(2) = 2* [C]
* *D({B, C}, C) = min( D({B}, B) + dB,C ) = min(4 + 6) = min(50) = 10* [B]
* *D({B, D}, B) = min( D({D}, D) + dD,B ) = min(9+ 5) = min(90) = 14* [D]
* *D({B, D}, D) = min( D({B}, B) + dB,D ) = min(4+ 11) = min(80) = 15* [B]
* *D({C, D}, C) = min( D({D}, D) + dD,C ) = min(9+(-4)) = min(80) = 5* [D]
* *D({C, D}, D) = min( D({C}, C) + dC,D ) = min(1+2) = min(3) = 3* [C]

Wyznaczamy wartości D(S, p) dla trójelementowych zbiorów S (s=3).

* *D({B, C, D}, B) = min( D({C, D}, C) + dC,B, D({3, D}, D) + dD,B ) = min(5 + 1, 3 + 5) = min(6, 8) = 6* [C]
* *D({B, C, D}, C) = min( D({B, D}, B) + dB,C, D({B, D}, D) + dD,C ) = min(14 +6, 15 +(-4)) = min(14, 11) = 11* [D]
* *D({B, C, D}, D) = min( D({B, C}, B) + dB,D, D({B, C}, C) + dC,D ) = min(2 + 11, 10 + 2) = min(13, 12) = 12* [C]

Trójelementowy zbiór S jest dla tego zadania największym z możliwych, gdyż nie licząc wierzchołka początkowego mamy trzy wierzchołki. Możemy więc już teraz wyznaczyć rozwiązanie całego zadania:

* *min( D({B, C, D}, B) + dB,A, D({B, C, D}, C) + dC,A, D({2, 3, 4}, 4) + d4,A,) = min(6 + 3, 11 + 4, 12 + 6) = min(9, 15, 16) = 9* [B]

Wyliczyliśmy koszt najkrótszej trasy. Ostateczną ścieżkę odtwarzamy, wykorzystując indeksy z kwadratowych nawiasów, w których zapisany jest przedostatni węzeł w ścieżce. Trasa to A–D–C–B–A.

#### Brach and Bound:

**Funkcja ograniczająca**

Zamieniamy zera na nieskończoności, aby algorytm redukcji macierzy przebiegł pomyślnie.

Redukujemy macierz, odszukując w wierszu najmniejszą wartość (która nie jest zerem) i odejmujemy ją od cyfr w danym wierszu. Tak samo postępujemy z kolumnami. Jeśli w wierszu/kolumnie występują same nieskończoności lub choć jedno 0 wiersz/kolumnę uznajemy za zredukowaną i pomijamy.

Następnie budujemy drzewo, w którym kolejne wierzchołki drzewa odpowiadają określonym wierzchołkom grafu. Dla wierzchołka drzewa 1 (odpowiadającemu wierzchołkowi grafu A) zredukujmy macierz – koszt, który wyniknie z obliczeń to zarazem globalne ograniczenie.

Macierz na drodze (rodzic, dziecko) kopiujemy od rodzica, a później aktualizujemy, zmieniając wartości w wierszu rodzica i kolumnie dziecka na nieskończoności. Nieskończoność należy również wpisać na skrzyżowaniu wiersza rodzica/dziadka/pradziadka itd. i kolumny dziecka w tworzonej macierzy.

1. Odpowiadający wierzchołek grafu: A

Nie można powiedzieć, że istnieje trasa o tej długości, ale wiemy, że nie istnieje trasa o długości mniejszej.

1. Odpowiadający wierzchołek grafu: B; droga A-B
2. Odpowiadający wierzchołek grafu: C; droga A-C

1. Odpowiadający wierzchołek grafu: D; droga A-D
2. Odpowiadający wierzchołek grafu: B; droga D-B

1. Odpowiadający wierzchołek grafu: C; droga D-C
2. Odpowiadający wierzchołek grafu: B; droga C-B

Obliczane koszty wpisywaliśmy w węzły drzewa odpowiadające podpunktom powyżej.

Node: 1

Koszt: 2

Node: 4

Koszt: 9

Node: 2

Koszt: 11

Node: 3

Koszt: 11

Node: 6

Koszt: 9

Node: 5

Koszt: 24

Node: 7

Koszt: 9

Sprawdzamy, czy koszt wierzchołka-liścia B, na którym skończyliśmy jest najmniejszy spośród jeszcze nie rozwiniętych wierzchołków. Inne koszty to kolejno, idąc wszerz: 11, 11, 24. 9 jest najmniejsza, a więc kończymy wykonywanie się algorytmu. Trasę odtworzymy, idąc po tych obiecujących wierzchołkach, które doprowadziły nas do liścia B o koszcie 9.

### Opis implementacji algorytmu

Graf pełny zapisywany jest w macierzy sąsiedztwa (nie istnieje połączenie z miasta do tego samego miasta, dlatego na przekątnej są zera), która jest wczytywana z pliku \*txt. W programie występuje klasa Graph, która trzyma w sobie informacje na temat grafu takie jak: liczba wierzchołków i macierz sąsiedztwa (tablica dwuwymiarowa alokowana dynamicznie). Oprócz tego zaimplementowane zostały 3 klasy – każda wykorzystywana do innego algorytmu: TSP\_BF, TSP\_BB i TSP\_DP.

Wykorzystywane struktury w klasach (oprócz matrix\_distance, czyli macierzy sąsiedztwa):

* **TSP\_BF** – tablice: permutacji, w której zwracane są kolejne permutacje ścieżki oraz finalny tor.
* **TSP\_BB** – kolejka priorytetowa, z której pobieramy wierzchołek o najmniejszym koszcie (obiecujący), vector gromadzący dzieci węzła
* **TSP\_DP** – tablica memory (omówiona w punkcie 2.1), vector na podzbiory z jedynkami umieszczonymi na wszystkich możliwych pozycjach oraz vector, w którym zapisywana jest ostateczna ścieżka.

Dla problemu BB nie są przyjmowane grafy o liczbie wierzchołków większej niż 16 (wyjątek powodujący wyświetlanie się ostrzeżenia).

### Plan eksperymentu

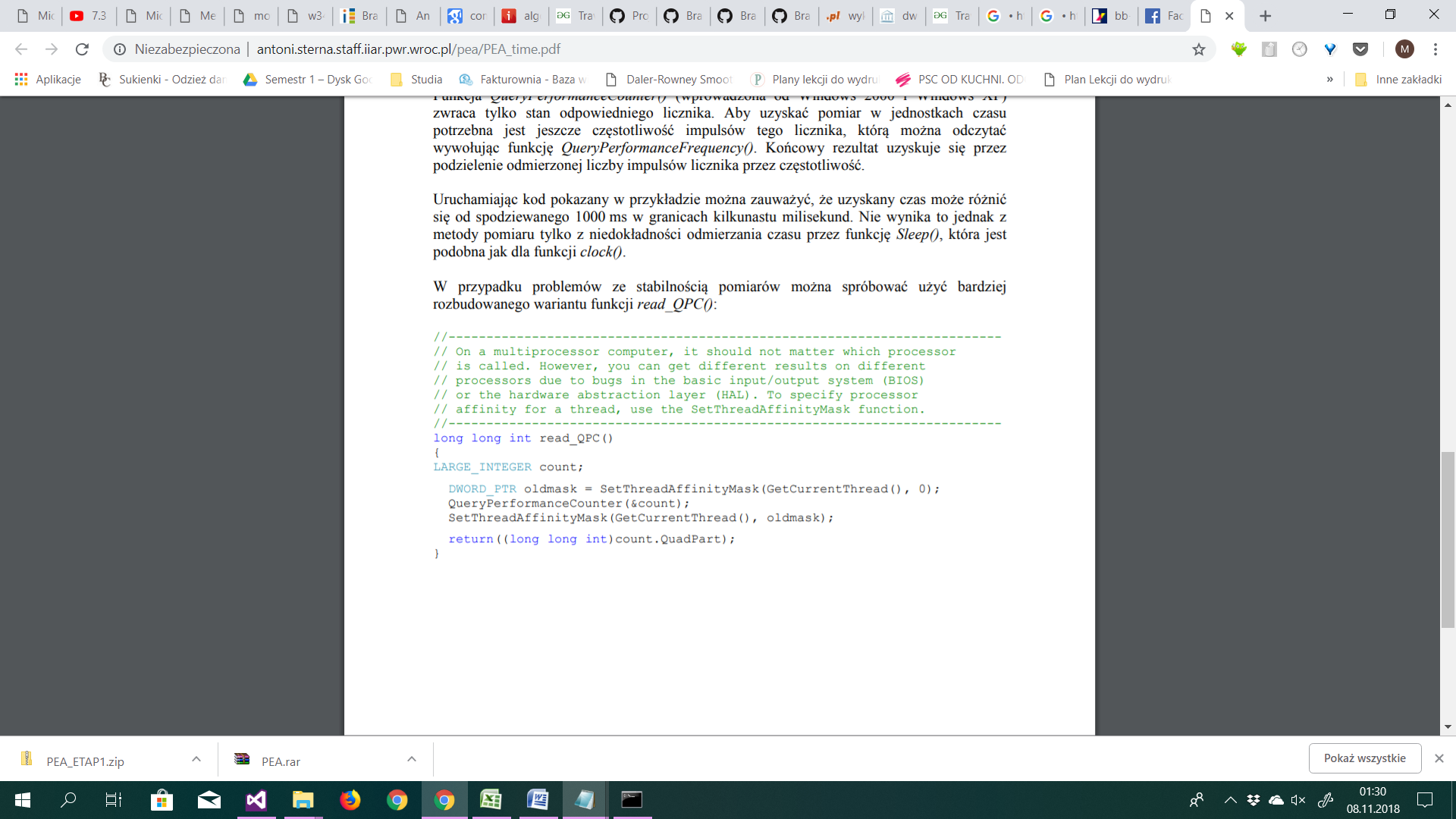
Program napisany w C++ ma zaimplementowane trzy algorytmy rozwiązujące problem TSP:

* Przeglądu zupełnego (BF - Brute Force)
* Podziału i ograniczeń (Branch and Bound – B&B)
* Programowania dynamicznego (DP – Dynamic Programming)

Przyjęłam następujące założenia:

* pierwsze miasto (indeks 0) przyjęto za punkt końcowy i początkowy podróży,
* dane na przekątnej macierzy wczytywanej z pliku są równe 0
* wagi krawędzi są liczbami całkowitymi typu integer
* używane struktury danych są alokowane dynamicznie (w zależności od aktualnego rozmiaru problemu),
* program umożliwia weryfikację poprawności działania algorytmu (wczytanie danych wejściowych z pliku tekstowego),
* po zaimplementowaniu i sprawdzeniu poprawności działania algorytmu przeprowadzono pomiar czasu jego działania w zależności od rozmiaru problemu N (badania wykonano dla 7 różnych reprezentatywnych wartości N),
* dla każdej wartości N wygenerowano po 100 losowych instancji problemu (wyniki uśrednione),
* implementacje algorytmów powinny być zgodne z obiektowym paradygmatem programowania.

Dane do testów generowane były losowo za pomocą funkcji rand. Pomiar czasu wykorzystywał funkcję QueryPerformanceCounter() ze strony <http://antoni.sterna.staff.iiar.pwr.wroc.pl/pea/PEA_time.pdf>



Stworzyłam funkcję liczącą permutacje na podstawie filmiku *Coding Interview Problem: Permutation Generator.* W skrócie polega on na tym, że znajdujemy w wektorze (wszystkich elementów) punkt, od którego wartości maleją, a potem odwracamy kolejność reszty wartości od tego punktu.

### Wyniki eksperymentów

Testy przeprowadzone zostały w trybie Release i polegały na wykonaniu pomiarów stukrotnych dla grafów o reprezentatywnej liczbie wierzchołków. Jako, że obliczenia dla mniejszych wartości trwały dość krótko, zrobiłam pomiary dla grafów o wagach generowanych losowo z przedziału -1000 do 1000 o następującej liczbie wierzchołków: {3,4,5,6,7,8,9,10,11,12,13} = zbiór W. Wykonanie pomiarów dla TSP powyżej 13 wierzchołków trwało zbyt długo, więc testy dla algorytmów B&B i Daynamic Programming zrobiłam również na zbiorze W, aby wykresy dla wszystkich algorytmów przedstawić na jednym wykresie.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| B&B | | | | | | | | | | | |
| Wierzchołki | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| Czas [us] | 65 | 108 | 169 | 271 | 531 | 1195 | 2451 | 4359 | 10822 | 27528 | 51633 |
|  |  |  |  |  |  |  |  |  |  |  |  |
| BF | | | | | | | | | | | |
| Wierzchołki | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| Czas [us] | 37 | 52 | 59 | 84 | 306 | 1964 | 16644 | 168793 | 1982331 | 24933527 | 149065000 |
|  |  |  |  |  |  |  |  |  |  |  |  |
| DP | | | | | | | | | | | |
| Wierzchołki | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| Czas [us] | 38 | 52 | 62 | 90 | 147 | 333 | 888 | 2119 | 4826 | 11380 | 24838 |

Przedstawmy wszystkie 3 algorytmy na wykresie o podziałce pionowej liniowej:

Na osi pionowej są sekundy, ponieważ tak czytelniej można określić wartości dla algorytmu BF, którego wyniki są na wykresie najbardziej widoczne.

Wykres 1 i 2: Przewaga obliczeniowa BF dla {12, 13} uniemożliwia odczytanie wartości dla algorytmów DP i B&B

Duża złożoność czasowa algorytmu BF dla {12, 13} uniemożliwia odczytanie wartości dla algorytmów DP i B&B.

W tym przypadku korzystniejsze będzie przedstawienie wykresów w skali logarytmicznej. Na osi pionowej są teraz mikrosekundy, gdyż w takich jednostkach wykonywane były pomiary.

Wykres 3: Wykres słupkowy logarytmiczny dla BF, DP i B&B

Wykres 4: Wykres liniowy logarytmiczny dla BF, DP i B&B

Wyniki dla algorytmu TSP ze względu na duże wartości zaburzają porównanie wyników dla DP i B&B. Poniżej przedstawiłam wykresy tylko dla tych dwóch algorytmów.

Wykres 5: Wykres liniowy DP i B&B

Przyjrzyjmy się również odcinkowi wykresu, na którym zobaczymy punkt przecięcia się wykresów DP i BF. W punkcie 1.3 zamieściłam wykresy funkcji złożoności czasowej dla BF i DP, korzystając z programu Wolfram Alpha. Wykres, który wyszedł mi z moich obliczeń zamieściłam poniżej.

Dla 15 wierzchołków czas jest dłuższy dla B&B (średnio 4543455 us) niż DP (356252 us), dlatego na wykresie nie przedstawiałam wyników dla n > 15.

Wykres 6: Wykres liniowy dla BF, DP i B&B dla {3,4,5,6,7}

### Wnioski

Pierwszym wnioskiem jaki nasuwa się w przypadku wszystkich trzech mówionych algorytmów jest fakt, że wzrost czasu wykonania algorytmu rośnie wprost proporcjonalnie do rozmiaru grafu.

Algorytm B&B okazał się pamięciowo obszerny, gdyż klasa węzła Node przechowuje dwuwymiarową tablicę macierzy indywidualną dla każdego węzła powstającą na podstawie macierzy rodzica. Tyle, ile węzłów, tyle powstaje macierzy 2D o rozmiarach NxN. Gdyby użyć innej funkcji ograniczającej i nie korzystać z informacji zawartych w macierzy rodzica, moglibyśmy wyeliminować obecność dwuwymiarowej tablicy w klasie Node, co zwiększyłoby szybkość wykonywania algorytmu.

Błędem, który okazał się trudny do zdiagnozowania, bo nie pojawiał się zawsze (tylko przy macierzach sąsiedztwa grafu, w których występowały zera nie tylko na przekątnej) był zły warunek w pewnym ifie, który zamieniał wszystkie zera (nie tylko na przekątnej) na nieskończoności. Ogólnie, taki zabieg zamiany zer na przekątnej na nieskończoności potrzebny jest do prawidłowego działania algorytmu (aby mogły być wyłapywane nieskończoności oraz zera wynikające z redukcji macierzy).

W dwóch pętlach for odbywało się przypisanie (A to wejściowa macierz sąsiedztwa grafu – oryginalna z zerami na przekątnej).

if (A[i][j] == 0) matrix\_distance[i][j] = INT\_MAX;

Należało je zmienić na takie:

if (i == j) matrix\_distance[i][j] = INT\_MAX;

Podczas pisania kodu, zauważyłam, że poszczególne algorytmy mogą wykorzystywać te same metody np. run(), która uruchamia metodę i display\_results(), która wyświetla wyniki. Moglibyśmy zrobić klasę wirtualną np. o nazwie TSP\_algorythm i wykorzystać polimorfizm. Nie jest to jednak zabieg niezbędny, bo projekt jest względnie mały, ale pozwala na dużą elastyczność. Jeszcze lepszym rozwiązaniem wydaje się być stworzenie jednej klasy np. TSP, w której poszczególne algorytmy byłyby metodami.

**Wnioski wyników wykresów**

Z wykresów wynika, że złożoność czasowa dla większych argumentów (od ok. 8) dla Brute Force jest większa o kilka rzędów wielkości w porównaniu z algorytmami B&B i DP, co zgadza się z założeniami. (z wykresami funkcji n!); od 7 wierzchołka następuje znacząca przewaga algorytmów B&B i DP nad BF. Ciekawym wydaje się fakt, że szybszy jest algorytm Brute Force dla n < 7.

Algorytm Dynamic Programming wypada lepiej niż Branch and Bound – choć różnica z poziomu wykresu dla TSP wierzchołków powyżej 8 wydaje się niezauważalna. Trudno rozpatrywać rezultat w kategorii poprawności, gdyż brakuje danych o złożoności algorytmu B&B – złożoność zależy od wielu czynników.

Jednakże, im większy rozmiar instancji problemu tym większa jest korzyść z wykorzystania metody Branch and Bound oraz Dynamic Programming. Jeśli zależy nam na szybszym, ale niekoniecznie dokładnym rozwiązaniu dla dużego rozmiaru grafu, warto wziąć wtedy pod uwagę algorytm Branch and Bound.

### Bibliografia

* <https://inf.ug.edu.pl/~hanna/grafy/04_kom_opt.pdf?fbclid=IwAR0nEC60rp-AW_0qaCuNGq9WaODmI4qRlNa243AmnereVm_xeeFMb1qoPlQ>
* <https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/>
* http://algorytmy.ency.pl/artykul/algorytm\_helda\_karpa
* <http://delta.mimuw.edu.pl/artykuly/delta0208/komiwojazer.pdf>
* <http://www.enseignement.polytechnique.fr/informatique/INF431/X09-2010-2011/AmphiLL/branch_and_bound_for_TSP-notes.pdf>
* <http://p.wi.pb.edu.pl/sites/default/files/krzysztof-ostrowski/files/w34.pdf>
* <http://www.intelligence.tuc.gr/~petrakis/courses/datastructures/algorithms.pdf>
* <https://www.google.pl/search?q=google+tlumacz&rlz=1C1PRFI_enPL712PL727&oq=google+tlumacz&aqs=chrome..69i57j0l5.2358j0j7&sourceid=chrome&ie=UTF-8>
* Algorytm Branch and Bound na podstawie: <https://www.youtube.com/watch?v=1FEP_sNb62k>
* Pomiar czasu: <http://antoni.sterna.staff.iiar.pwr.wroc.pl/pea/PEA_time.pdf>
* T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Wprowadzenie do algorytmów, Wydawnictwo Naukowe PWN, 2013
* Coding Interview Problem: Permutation Generator https://www.youtube.com/watch?v=V7hHupttzVk

[dostęp dnia: 7.11.2018]