

Projektowanie efektywnych algorytmów

Projekt 1

Implementacja algorytmów dokładnych dla problemu komiwojażera

Autor: Mateusz Chalik

Nr indeksu: 252735

Termin zajęć: Czwartek 15.15

Prowadzący: Mgr inż. Antoni Sterna

Spis treści

1. Wstęp teoretyczny	2
2. Praktyczny opis działania algorytmów	3
3. Opis implementacji algorytmu	8
4. Plan eksperymentu.....	11
5. Wyniki eksperymentów.....	13
6. Wnioski	15

1. Wstęp teoretyczny

1.1 Opis problemu

TSP (ang. travelling salesman problem) - problem komiwojażera

Nazwa pochodzi od typowej ilustracji problemu, przedstawiającej go z punktu widzenia wędrownego sprzedawcy (komiwojażera): dane jest n miast, które komiwojażer ma odwiedzić, oraz odległość pomiędzy każdą parą miast. Celem jest znalezienie właściwego cyklu Hamiltona - najkrótszej drogi łączącej wszystkie miasta, zaczynającej się i kończącej się w określonym punkcie.

Symetryczny problem komiwojażera (STSP) polega na tym, że dla dowolnych miast A i B odległość z A do B jest taka sama jak z B do A. W asymetrycznym problemie komiwojażera (ATSP) odległości te mogą być różne. Główną trudnością problemu jest duża liczba danych do analizy.

1.2 Opis algorytmów

Algorytm przeglądu zupełnego - gwarantuje optymalne wyniki ale jest bardzo wolny, polega na permutacjach - sprawdza wszystkie możliwe ścieżki poprzez sprawdzenie wszystkich kombinacji podanych wierzchołków / miast.

Algorytm podziału i ograniczeń - metoda opiera się na przeszukiwaniu drzewa reprezentującego przestrzeń rozwiązań problemu, odcięcia redukują liczbę przeszukiwanych wierzchołków przyspieszając rozwiązanie. W danej metodzie rozgałęzienia tworzą następników danego wierzchołka a ograniczenia odcinają części drzewa, w których na pewno nie ma optymalnego rozwiązania.

Algorytm programowania dynamicznego - idea programowania dynamicznego polega na zapisywaniu w pamięci danych z aktualnych podwywołań używając rekurencji. Kolejne podwywołania prowadzą do coraz to łatwiejszych problemów i tak aż to trywialnych.

1.3 Szacowana złożoność obliczeniowa

Przegląd zupełny: $O(n!)$

Metoda podziału i ograniczeń: *wykładnicza*, $O(2^n)$

Metoda programowania dynamicznego: $O(n^2 2^n)$

2. Praktyczny opis działania algorytmów

2.1 Algorytm przeglądu zupełnego

	0	1	2	3
0	0	2	7	6
1	4	0	8	5
2	5	3	0	1
3	9	4	8	0

1) Wartości początkowe

min = INT_MAX
path = null

2) Obliczamy koszt każdej możliwej ścieżki:

$\{0\} \{1, 2, 3\} = 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0 = 27$	min = 27	path = {0, 1, 2, 3, 0}
$\{0\} \{1, 3, 2\} = 0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 0 = 27$	min = 27	path = {0, 1, 2, 3, 0}
$\{0\} \{2, 1, 3\} = 0 \rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 0 = 24$	min = 24	path = {0, 2, 1, 3, 0}
$\{0\} \{2, 3, 1\} = 0 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 0 = 16$	min = 16	path = {0, 2, 3, 1, 0}
$\{0\} \{3, 1, 2\} = 0 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 0 = 23$	min = 16	path = {0, 2, 3, 1, 0}
$\{0\} \{3, 2, 1\} = 0 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 0 = 21$	min = 16	path = {0, 2, 3, 1, 0}

2.2 Algorytm podziału i ograniczeń

	0	1	2	3
0	0	9	7	6
1	4	0	8	5
2	5	3	0	1
3	9	4	8	0

1) Wartości początkowe

```
lowerBound = 0
upperBound = INT_MAX
path = null
cost = 0
bestCost = INT_MAX
possiblePath = null
visited = [false, false, false, false]
```

2) Obliczamy początkowe dolne ograniczenie dodając minimum z każdego wiersza

```
lowerBound = 6 + 4 + 1 + 4 = 15
possiblePath = {0}
visited = [true, false, false, false]
```

3) Przeszukiwanie drzewa (lowerBound = 15, cost = 0, level = 1)

(1) rekurencja

```
level != 4
przeszukujemy nieodwiedzone wierzchołki z wiersza o indeksie [level - 1]
i = 1, bierzemy wierzchołek o indeksach [0][1]
poprzednie lowerBound zapisujemy do tymczasowej zmiennej, lowerBound = 0
cost = 0 + 9 = 9
dodanie do lowerBound minimów z wierszy, których indeksy w visited są równe false
lowerBound = 9
lowerBound + cost < bestCost : possiblePath = {0, 1}
visited = [true, true, false, false]
```

przeszukiwanie drzewa (lowerBound = 9, cost = 9, level = 2)

(2) rekurencja

```
level != 4
przeszukujemy nieodwiedzone wierzchołki z wiersza o indeksie [level - 1]
i = 2, bierzemy wierzchołek o indeksach [1][2]
poprzednie lowerBound zapisujemy do tymczasowej zmiennej, lowerBound = 0
cost = 9 + 8 = 17
dodanie do lowerBound minimów z wierszy, których indeksy w visited są równe false
lowerBound = 8
lowerBound + cost < bestCost : possiblePath = {0, 1, 2}
visited = [true, true, true, false]
```

przeszukiwanie drzewa (lowerBound = 8, cost = 17, level = 3)

(3) rekurencja

level != 4

przeszukujemy nieodwiedzone wierzchołki z wiersza o indeksie [level - 1]

i = 3, bierzemy wierzchołek o indeksach [2][3]

poprzednie lowerBound zapisujemy do tymczasowej zmiennej, lowerBound = 0

cost = 17 + 1 = 18

dodanie do lowerBound minimów z wierszy, których indeksy w visited są równe false

lowerBound = 4

lowerBound + cost < bestCost : possiblePath = {0, 1, 2, 3}

visited = [true, true, true, true]

przeszukiwanie drzewa (lowerBound = 4, cost = 18, level = 4)

(4) rekurencja

level == 4

dodanie połączenia pomiędzy wierzchołkiem possiblePath[level - 1] a possiblePath[0]

result = 18 + 9 = 27

result < bestCost : bestCost = 27, path = {0, 1, 2, 3, 0}

znaleziono nową optymalną ścieżkę oraz koszt

(5) wyjście z rekurencji

level = 2, i = 2

odjęcie kosztu pomiędzy wierzchołkiem possiblePath[level - 1] a possiblePath[i]

cost = 18 - 1 - 8 = 9

przypisanie do lowerBound zachowanej wartości: lowerBound = 9

visited = [false, false, false, false]

ustawienie visited na true do aktualnego poziomu: visited = [true, true, false, false]

i = 3, bierzemy wierzchołek o indeksach [1][3]

poprzednie lowerBound zapisujemy do tymczasowej zmiennej, lowerBound = 0

cost = 9 + 5 = 14

dodanie do lowerBound minimów z wierszy, których indeksy w visited są równe false

lowerBound = 5

lowerBound + cost < bestCost : possiblePath = {0, 1, 3}

visited = [true, true, false, true]

przeszukiwanie drzewa (lowerBound = 5, cost = 14, level = 3)

(6) rekurencja

level != 4

przeszukujemy nieodwiedzone wierzchołki z wiersza o indeksie [level - 1]

i = 2, bierzemy wierzchołek o indeksach [3][2]

poprzednie lowerBound zapisujemy do tymczasowej zmiennej, lowerBound = 0

cost = 14 + 8 = 22

dodanie do lowerBound minimów z wierszy, których indeksy w visited są równe false

lowerBound = 5

lowerBound + cost < bestCost : possiblePath = {0, 1, 3, 2}

visited = [true, true, true, true]

(7) rekurencja

level == 4

dodanie połączenia pomiędzy wierzchołkiem possiblePath[level - 1] a possiblePath[0]

result = 22 + 5 = 27

result !< bestCost : bestCost = 27, path = {0, 1, 2, 3, 0}

koszt nowej ścieżki jest taki sam jak poprzedni, nie zmieniamy optymalnego rozwiązania

itd...

2.3 Algorytm programowania dynamicznego

	0	1	2	3
0	0	9	7	6
1	4	0	8	5
2	5	3	0	1
3	9	4	8	0

1) Wartości początkowe

min = INT_MAX

bitMask = 0

newSubset = 0

counter = 1

costTable = inf

pathTable = inf

2) Szukanie minimum

Start rekurencji: (source = 0, set = $2^{\text{liczba miast}} - 1 - 2^{\text{source}} = 14$, bitMask = 0, newSubset = 0)

i = 0

bitMask = $2^{\text{liczba miast}} - 1 - 2^i = 16 - 1 - 1 = 14$

newSubset = set & bitMask = 14 AND 14 = 14

set = newSubset -> stop, kolejna iteracja

i = 1

bitMask = $16 - 1 - 2 = 13$

newSubset = set & bitMask = 14 AND 13 = 12

set != newSubset -> tempMin = matrix[0][1] + rekurencja(source = i, set = newSubset) = 9 + ...

i = 1

bitMask = 16 - 1 - 2 = 13

newSubset = set & bitMask = 12 AND 13 = 12

set = newSubset -> stop, kolejna iteracja

i = 2

bitMask = 16 - 1 - 4 = 11

newSubset = set & bitMask = 12 AND 11 = 8

set != newSubset -> tempMin = matrix[1][2] + rekurencja(source = i, set = newSubset) = 8 + ...

i = 2

bitMask = 16 - 1 - 4 = 11

newSubset = set & bitMask = 8 AND 11 = 8

set = newSubset -> stop, kolejna iteracja

i = 3

bitMask = 16 - 1 - 8 = 7

newSubset = set & bitMask = 8 AND 7 = 0

set != newSubset -> tempMin = matrix[2][3] + rekurencja(source = i, set = newSubset) = 1 + ...

i = 3

bitMask = 16 - 1 - 8 = 7

newSubset = set & bitMask = 0 AND 7 = 0

set != newSubset -> koniec iteracji

itd...

3) Końcowe zapisane tablice oraz odczyt rozwiązania:

costTable:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	16	-1
1	4	-1	-1	-1	13	-1	-1	-1	14	-1	-1	-1	18	-1	-1	-1
2	5	-1	7	-1	-1	-1	-1	-1	10	-1	9	-1	-1	-1	-1	-1
3	9	-1	8	-1	13	-1	15	-1	-1	-1	-1	-1	-1	-1	-1	-1

początkowa wartość set = $2^{\text{liczba miast}} - 1 - 2^{\text{source}} = 14$

cost = costTable [0][14] = 16

pathTable:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	2	-1
1	-1	-1	-1	-1	2	-1	-1	-1	3	-1	-1	-1	2	-1	-1	-1
2	-1	-1	1	-1	-1	-1	-1	-1	3	-1	3	-1	-1	-1	-1	-1
3	-1	-1	1	-1	2	-1	2	-1	-1	-1	-1	-1	-1	-1	-1	-1

początkowa wartość set = $2^{\text{liczba miast}} - 1 - 2^{\text{source}} = 14$

start = 0, counter = 1, set = 14 -> path[counter] = pathTable[start][set] -> path[1] = 2

start = 2, counter = 2, set = 10 -> path[2] = pathTable [2][10] = 3

start = 3, counter = 3, set = 2 -> path[3] = pathTable [3][2] = 1

path = {0, 2, 3, 1, 0}

3. Opis implementacji algorytmu

Graf został zaimplementowany jako osobna klasa posiadająca różne metody. Koszt pomiędzy wierzchołkami są zaimplementowane jako wektor dwuwymiarowy typu int a ścieżka jako tablica typu int.

Do metod sterującymi algorytmami podajemy macierz oraz pustą tablicę, która zostanie wypełniona przez algorytm optymalną ścieżką. Metody zwracają koszt optymalnej ścieżki.

3.1 Algorytm przeglądu zupełnego

Główna pętla programu:

```
do {  
    // obliczamy ścieżkę  
    int currentCost = calculate(nodes, matrix);  
  
    // jeżeli koszt jest mniejszy od dotychczasowego  
    if (currentCost < minCost) {  
  
        // znaleziono nową najkrótszą ścieżkę  
        minCost = currentCost;  
  
        // przypisanie ścieżki  
        for (int i = 0; i < matrix.size(); ++i) {  
            bestPath[i] = nodes[i];  
        }  
    }  
  
    // sprawdzenie wszystkich możliwych kombinacji  
} while (next_permutation(nodes, nodes + matrix.size()));
```


Gdzie funkcja *calculate* liczy ścieżkę i koszt podanej kombinacji wierzchołków.

Funkcja *next_permutation* oblicza wszystkie kombinacje podanego zbioru wierzchołków.

3.2 Algorytm podziału i ograniczeń

Funkcja obliczająca minimum w danym wierszu.

```
int BranchAndBound::minimumLine(int line) {  
  
    int cost = INT_MAX;  
  
    // przeszukujemy cały wiersz  
    for (int i = 0; i < matrixSize; i++) {  
  
        // jeżeli znaleziono minimum i wierzchołek do którego prowadzi  
        // minimum nie został jeszcze odwiedzony  
        if (thisMatrix[line][i] < cost && line != i && visited[i] == false)  
  
            // przypisujemy minimum  
            cost = thisMatrix[line][i];  
    }  
  
    return cost;  
}
```

Górne ograniczenie zostaje nadpisane, gdy podczas przeszukiwania drzewa dotrzemy do liścia.

```
// gdy dotarliśmy do ostatniego poziomu (SCHODZENIE W GŁĄB AŻ DO LIŚCIA)  
if (level == matrixSize) {  
  
    // jeżeli istnieje połączenie  
    if (thisMatrix[possiblePath[level - 1]][possiblePath[0]] != -1) {  
  
        int result = cost + thisMatrix[possiblePath[level -  
1]][possiblePath[0]];  
  
        // znaleziono nowe optymalne rozwiązanie  
        if (result < bestCost) {  
  
            // przepisanie ścieżki  
            for (int i = 0; i < matrixSize; i++)  
                path[i] = possiblePath[i];  
  
            // dodanie pierwszego miasta na koniec  
            path[matrixSize] = possiblePath[0];  
  
            // optymalny koszt  
            bestCost = result;  
        }  
    }  
    return;  
}
```

Z racji tego, że jest to schodzenie w głąb, jest tworzone wiele dolnych ograniczeń, ich wartości są zapamiętywane na danym poziomie a nowe dolne ograniczenie jest liczone podczas kolejnego wywołania rekurencji na poziomie wyższym.

```
// tymczasowe dolne ograniczenie (DAJE ZA KAŻDYM RAZEM NOWE DOLNE
// OGRANICZENIE)
int tempBound = lowerBound;
lowerBound = 0;

// dodanie kosztu
cost += thisMatrix[possiblePath[level - 1]][i];

// dla wszystkich węzłów
for (int j = 1; j < matrixSize; j++) {

    // resetowanie tablicy z odwiedzionymi wierzchołkami
    if (j == i)
        visited[0] = true;
    else
        visited[0] = false;

    // dodanie do dolnego ograniczenia minimów z lini (potomków których nie
    // odwiedziono)
    if (visited[j] == false)
        lowerBound += minimumLine(j);
}
```

Początkowe ustalenie dolnej i górnej granicy.

```
// górna granica
bestCost = INT_MAX;

// obliczanie dolnej granicy
for (int i = 0; i < matrixSize; i++)
    lowerBound += minimumLine(i);
```

3.3 Algorytm programowania dynamicznego

Głównym elementem jest utworzenie pomocniczych tablic. W jednej są przechowywane minima w zakresie podwywołań a na drugiej (w tych samych miejscach) wierzchołki.

```
for (int i = 0; i < matrixSize; i++) {

    // uzupełnianie tablicy 2d wartościami początkowymi (-1/inf)
    for (int j = 0; j < pow(2, matrixSize); j++) {
        costTable[i].push_back(-1);
        pathTable[i].push_back(-1);
    }
}
```

Główną częścią rekurencji jest wyliczanie nowych masek oraz podzbiorów, dalej zapisywanie ich we wcześniej utworzonych tablicach.

Maksa bitowa jest wyliczana ze wzoru: $2^{\text{liczba miast} - 1} - 2^{\text{aktualny wierzchołek}}$.

Podzbiór jest wyliczany przy pomocy funkcji logicznej *AND* na bitach, które reprezentują zbiór.

```
// wyliczenie nowej maski bitowej
bitMask = (int)((1 << matrixSize) - 1 - (1 << i))

// przypisanie nowego podzbioru (użycie bitowego AND)
newSubset = (int)set & bitMask;

// jeżeli aktualny zbiór jest inny od nowego
if (newSubset != set) {

    // wyliczenie kosztu
    tempMin = matrix[source][i] + findMinimum(i, newSubset, matrix);
}
```

4. Plan eksperymentu

Dla przeglądu zupełnego wielkości instancji wynosiły: 5, 6, 7, 8, 9, 10, 11, 12 wierzchołków.

Dla metody podziału i ograniczeń oraz programowania dynamicznego wielkości instancji wynosiły: 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17 i 18 wierzchołków.

Algorytmy zostały wywołane 100 razy a wyniki dla każdego przypadku uśredniano.

Aby pomiary były rzetelne, przed każdym wywołaniem algorytmu generowany jest nowy zestaw danych.

Pomiary są w pełni automatyczne, zostały zapisane do pliku a dalej analizowane.

Szybka funkcja generująca liczby pseudolosowe:

```
int Randomize::random_mt19937(int min, int max) {

    static uniform_int_distribution<int> uid(min,max);
    return uid(rng);
}
```

Sposób generowania danych poprzez konstruktor klasy Graph:

```
Graph::Graph(int size) {  
  
    // losowanie  
    Randomize r;  
  
    // rezerwujemy pamięć  
    matrix.reserve(size);  
  
    for (int i = 0; i < size; i++) {  
        vector<int> temp;  
        temp.reserve(size);  
  
        for (int j = 0; j < size; j++) {  
            temp.push_back(r.random_mt19937(1, 99));  
        }  
  
        temp[i] = 0;  
        this->matrix.push_back(temp);  
    }  
  
    // uzupełnianie przekątnej zerami  
    for (int i = 0; i < size; i++) {  
        matrix[i][i] = 0;  
    }  
}
```

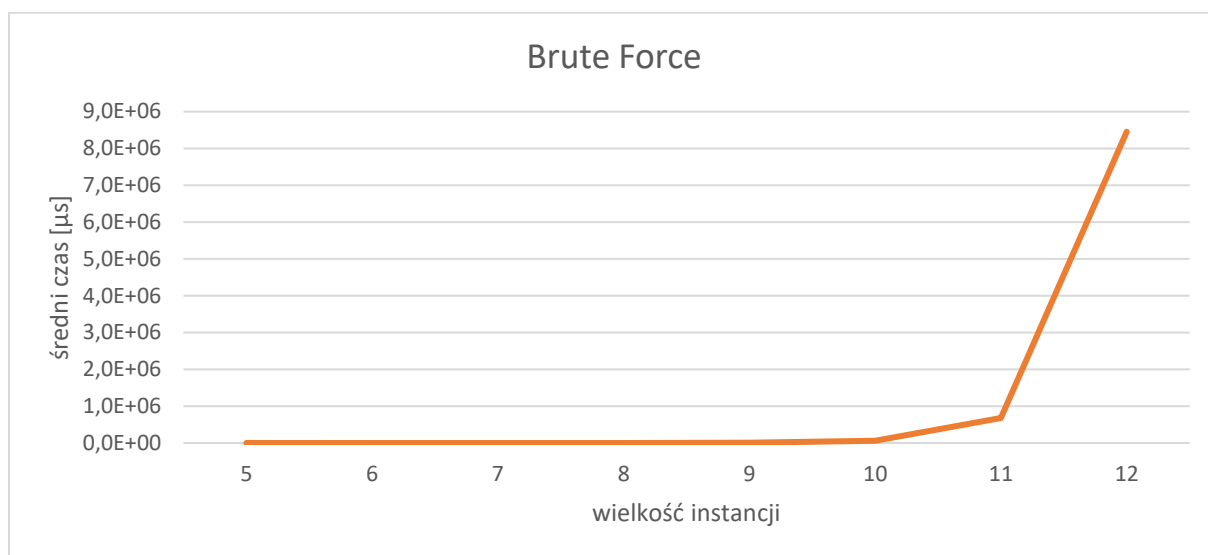
Sposób mierzenia czasu:

```
for (int i = 0; i < reps; ++i) {  
  
    graph = new Graph(instanceSize);  
  
    bb = new BranchAndBound();  
    path = new int[instanceSize + 1];  
  
    QueryPerformanceFrequency((LARGE_INTEGER *)&frequency);  
    start = read_QPC();  
  
    bb->algorithmBranchAndBound(graph->getMatrix(), path);  
  
    elapsed = read_QPC() - start;  
    sum += (1000000.0 * elapsed) / frequency;  
}  
file << "B&B: " << instanceSize << " miast,  czas [us]:  suma: " <<  
setprecision(0) << sum << ",  średnia: " << setprecision(0) << sum / reps << endl;  
sum = 0;
```

5. Wyniki eksperymentów

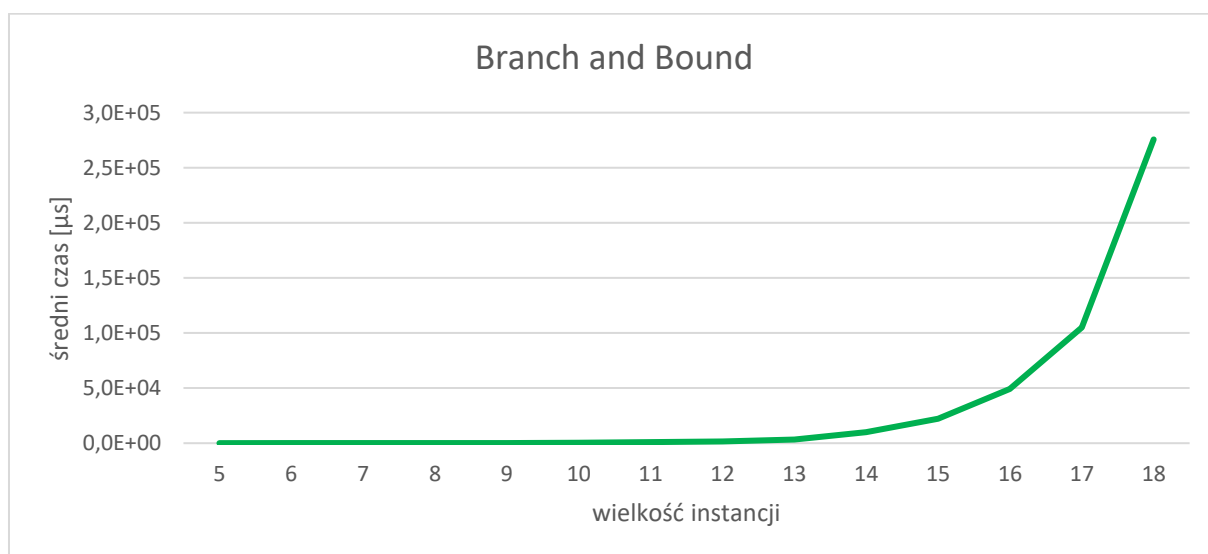
5.1 Algorytm przeglądu zupełnego

wielkość instancji	5	6	7	8	9	10	11	12
średni czas [μ s]	32	47	115	612	5284	59403	677946	8454398



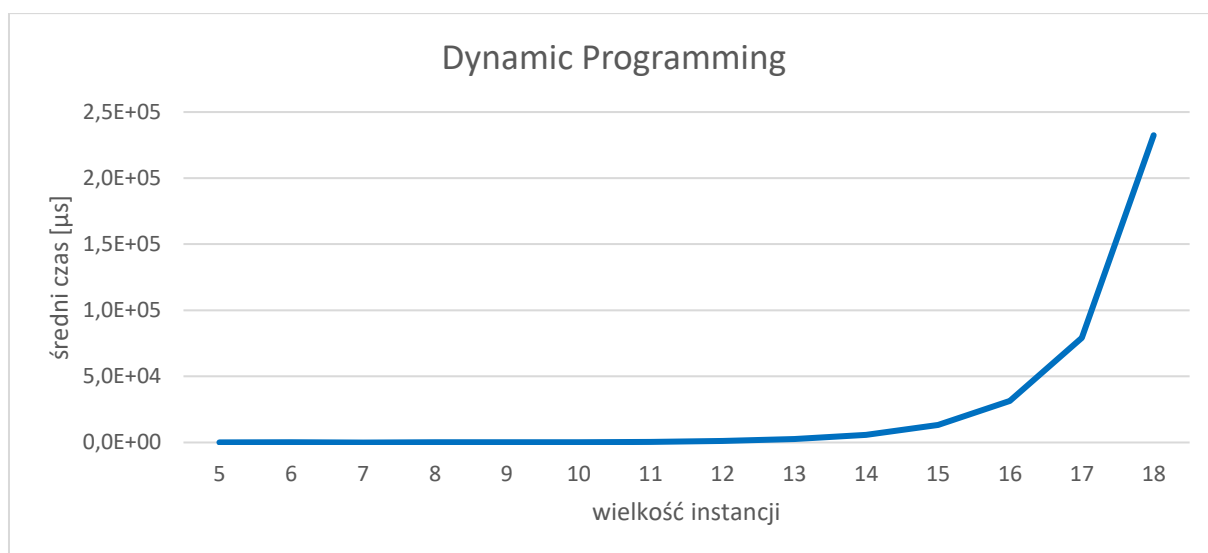
5.2 Algorytm podziału i ograniczeń

wielkość instancji	5	6	7	8	9	10	11	12	13	14	15	16	17	18
średni czas [μ s]	35	44	46	80	159	358	853	1477	3421	9972	22147	49276	104966	275692

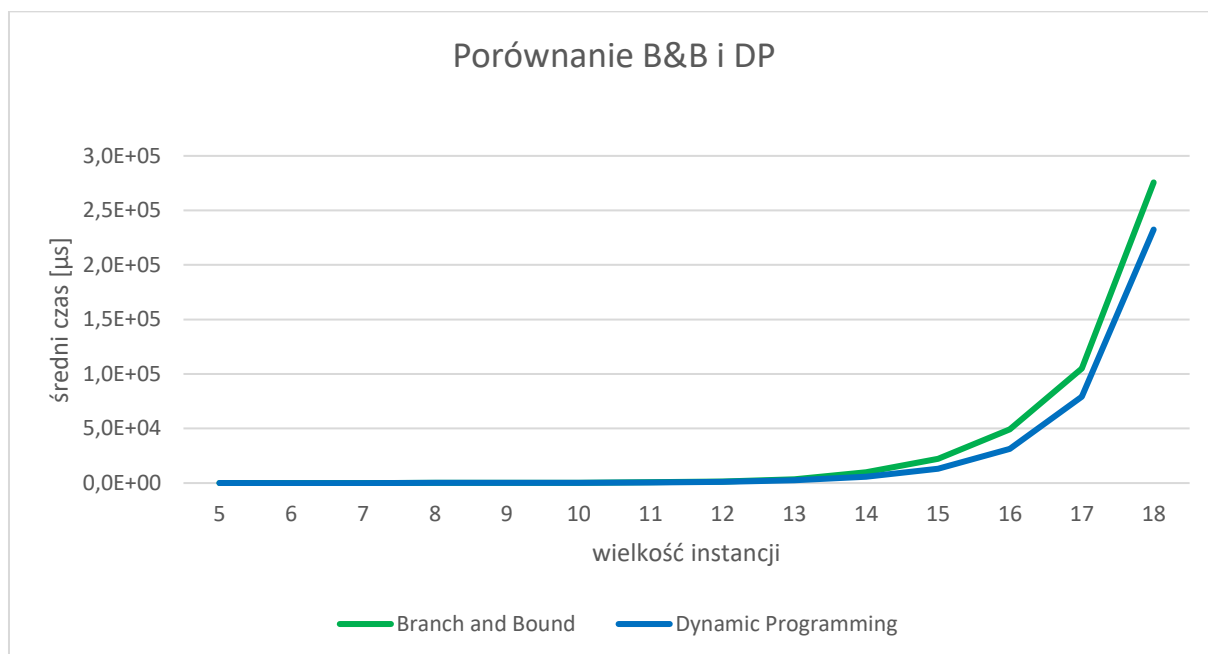


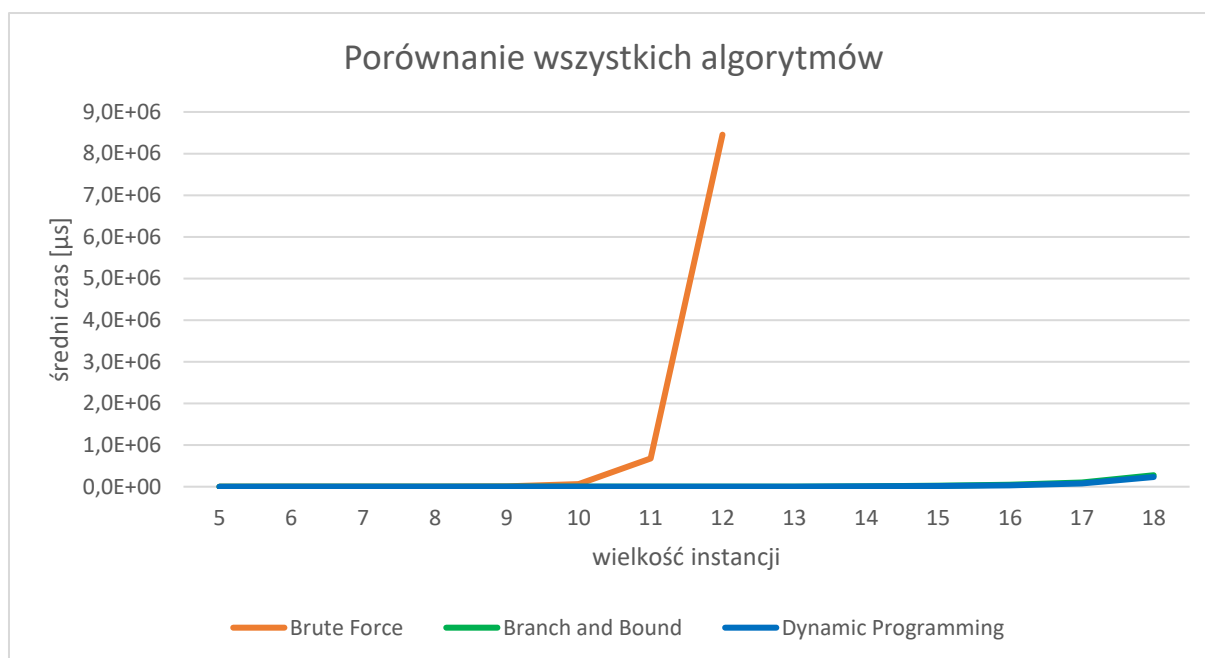
5.3 Algorytm programowania dynamicznego

wielkość instancji	5	6	7	8	9	10	11	12	13	14	15	16	17	18
średni czas [μ s]	50	65	58	92	137	253	522	1201	2592	5753	13164	31353	79140	232468



5.4 Porównanie algorytmów





6. Wnioski

Pomiary wykazały, że najszybszym algorytmem spośród badanych jest algorytm programowania dynamicznego. Metoda podziału i ograniczeń dla większych struktur okazała się nieznacznie gorsza od wyżej wymienionej metody, co obrazuje wykres porównania tych dwóch algorytmów. Jednakże, algorytm podziału i ograniczeń jest bardzo zależny od danych. Rozbieżności pomiędzy pomiarami są bardzo duże a końcowy uśredniony wynik dla różnych pomiarów może się znacząco różnić.

Najwolniejszym algorytmem jest przegląd zupełny. Gwarantuje on optymalny wynik, lecz jest bardzo wolny, co doskonale przedstawia wykres porównania wszystkich algorytmów.

Wykresy poszczególnych algorytmów pokazują, że każdy z nich ma wykładniczą złożoność obliczeniową. Porównując wszystkie trzy algorytmy zauważamy jednak, że złożoność obliczeniowa przeglądu zupełnego jest zdecydowanie mniej korzystna.

Początkowa wersja algorytmu przeglądu zupełnego była bardzo wolna. To przez przekazywanie macierzy jako argumentu do funkcji obliczającej ścieżkę z danej kombinacji wierzchołków. Utworzenie macierzy globalnej i branie z niej wartości zamiast przekazywania jej jako argument zdecydowanie przyspieszyło algorytm.

Udała się również optymalizacja początkowej wersji algorytmu programowania dynamicznego. Tak jak w przypadku przeglądu zupełnego utworzenie globalnej macierzy zamiast przekazywania jej jako parametr funkcji oraz przekazywanie do funkcji pojedynczych zmiennych zamiast tworzenie ich jako globalne przyspieszyło lekko algorytm. Natomiast kluczową rolę w optymalizacji algorytmu okazała się zmiana obliczania potęgi, ze zwykłego $\text{Math.pow}(2, x)$ na przesunięcie bitowe $1 \ll x$.