Sprawozdanie – Projektowanie Efektywnych Algorytmów – Projekt

Projekt nr 1 – Implementacja i analiza efektywności algorytmu przeszukiwania zupełnego oraz podziału i ograniczeń

Autor:	Adam Czekalski
Nr indeksu:	264488
Termin:	Piątek, godz. 13:15 – 15:00
Prowadzący:	dr inż. Jarosław Mierzwa

1. Wstęp teoretyczny

Problem komiwojażera opiera się na tym, że dany jest sprzedawca, który musi odwiedzić N miast. Zakładamy, że istnieje droga pomiędzy każdym miastem (jest to graf pełny o N(N-1) drogach). Każda droga ma swoją odległość (wagę). W przypadku symetrycznego problemu komiwojażera, odległość między danymi miastami jest taka sama w obu kierunkach. Jednak w tym przypadku rozważany jest asymetryczny problem komiwojażera, który różni się tym, że odległość w jednym kierunku do danego miasta niekoniecznie musi być taka sama jak w kierunku przeciwnym. Komiwojażer musi odwiedzić każde miasto dokładnie jeden raz, co oznacza, że utworzona droga przez komiwojażera musi być cyklem Hamiltona. Oprócz tego musi to zrobić jak najmniejszym kosztem.

2. Złożoność obliczeniowa

Złożoność obliczeniowa problemu TSP zależy głównie od metody jego rozwiązywania. W projekcie zostały rozważone dwa sposoby:

- Brute – force

  Jest to najbardziej naiwny sposób na rozwiązanie tego problemu. Polega na sprawdzeniu wszystkich możliwych permutacji dróg, nie stosując przy tym żadnych ograniczeń. W efekcie, algorytm jest efektywny tylko przy małej ilości miast, wraz ze wzrostem liczby miast N, czas wykonania algorytmu drastycznie wzrasta. Dla danego N, trzeba rozważyć (N-1)! możliwych rozwiązań (nie permutując wierzchołka startowego), z czego można oszacować, że złożoność rozwiązania TSP tym sposobem wynosi O(n!).

- Branch & Bound

  Ten sposób opiera się na liczeniu ograniczeń. Pozwala to na przerwanie liczenia kosztu ścieżki, jeśli w trakcie liczenia jej okaże się że na pewno nie będzie bardziej optymalna od tej obliczonej poprzednio. Jednak mimo wszystko, w najgorszym wypadku trzeba rozważyć wszystkie możliwe permutacje, czyli podobnie jak w przypadku metody brute – force złożoność wynosi O(n!).

3. Sposób działania algorytmów z ewentualnymi przykładami

- Brute – force

  Na początku tworzona jest tablica o rozmiarze N, która przechowuje ciąg arytmetyczny liczb z zakresu [0; N-1] o różnicy 1. Następnie poprzez rekurencję wyznaczane są kolejne możliwe permutacje tego ciągu. Np. dla N=4:
  Ciąg początkowy: 0123

Zamiana 2 ostatnich elementów: 0132
Powrót: 0123
Zamiana 1. elementu tablicy z 2. elementem: 0213
Zamiana 2 ostatnich elementów: 0231
Powrót: 0123
Zamiana 1. elementu tablicy z 3. elementem: 0321
Zamiana 2 ostatnich elementów: 0312
Powrót: 0123

Dla każdej możliwej permutacji liczona jest droga i jeśli droga aktualnej permutacji wierzchołków jest mniejsza od uprzednio najlepszej drogi, zmieniamy ciąg wierzchołków rozwiązania oraz najlepszą wartość drogi.

- Branch & Bound
    - Obliczanie ograniczeń:

```
int BranchAndBound::reduceMatrix(Matrix* matrix){
    //reducing matrix
    int min; //min value
    int reductionCost = 0;

    //reducing rows
    for (int i = 0; i < matrix->nrV; ++i) {

        min = INT_MAX;

        //find minimum value in a row
        for (int j = 0; j < matrix->nrV; ++j) {
            if(matrix->adjMatrix[i][j]<min && matrix->adjMatrix[i][j]!=-1) min = matrix->adjMatrix[i][j];
        }

        if(min>0 && min<INT_MAX){
            //subtract "min" from every value in that row
            reductionCost+=min;
            for (int j = 0; j < matrix->nrV; ++j) {
                if(matrix->adjMatrix[i][j]!=-1) matrix->adjMatrix[i][j] = matrix->adjMatrix[i][j] - min;
            }
        }
    }

    //reducing columns
    for (int j = 0; j < matrix->nrV; ++j) {

        min = INT_MAX;

        for (int i = 0; i < matrix->nrV; ++i) {
            if(matrix->adjMatrix[i][j]<min && matrix->adjMatrix[i][j]!=-1) min = matrix->adjMatrix[i][j];
        }

        if(min>0 && min<INT_MAX){
            reductionCost+=min;
            for (int i = 0; i < matrix->nrV; ++i) {
                if(matrix->adjMatrix[i][j]!=-1) matrix->adjMatrix[i][j] = matrix->adjMatrix[i][j] - min;
            }
        }
    }
    return reductionCost;
}
```

Rysunek 3-1 Metoda reduceMatrix() redukująca macierz

W pierwszej pętli następuje redukcja rzędów macierzy. Dla każdego rzędu iterujemy przez znajdujące się w nim elementy i szukamy w nim minimum (wyłączając komórki z wartością nieskończoność (-1)). Jeśli znaleziona minimalna wartość jest większa od 0 i mniejsza od nieskończoności (innymi słowy, jeśli znaleziono w danym rzędzie wartość różną od 0 i -1), to dodajemy ją do całkowitego kosztu redukcji macierzy oraz odejmujemy ją od wartości każdej komórki macierzy w danym rzędzie (wyłączając komórki z wartością nieskończoność (-1)). Podobną czynność wykonujemy dla kolumn macierzy. Dla każdej kolumny iterujemy przez znajdujące się w niej elementy i szukamy w niej minimum (wyłączając komórki z wartością nieskończoność (-1)). Jeśli znaleziona minimalna wartość jest większa od 0 i mniejsza od nieskończoności (innymi słowy, jeśli znaleziono w danej kolumnie wartość różną od 0 i -1), to dodajemy ją do całkowitego kosztu redukcji macierzy oraz odejmujemy ją od wartości każdej komórki macierzy w danej kolumnie (wyłączając komórki z wartością nieskończoność (-1)). Na koniec zwracany jest całkowity koszt redukcji macierzy.

W algorytmie, dla wierzchołka początkowego dolne ograniczenie jest obliczane poprzez redukcję macierzy wejściowej, wywołując metodę *reduceMatrix()*. Natomiast w przypadku dalszej części algorytmu:

```
100                 //1. reduce size
101                 for (int j = 0; j < matrix->nrV; ++j) { //filling a row and column with -1
102                     //if we analyze edge (1,2), then we fill row 1. with -1 and column 2. with -1
103                     node->matrix->adjMatrix[j][node->nrV] = -1;
104                     node->matrix->adjMatrix[prevNode->nrV][j] = -1;
105                 }
106
107                 //2. prevent from cycles
108                 node->matrix->adjMatrix[i][0] = -1; //we don't come back to the beginning verticle (nr 0)
109
110 //                std::cout << "Before reduction:" << std::endl;
111 //                node->matrix->printMatrix();
112
113                 //3. reduce matrix
114                 reduction = BranchAndBound::reduceMatrix(node->matrix); //reduce matrix
115
116 //                std::cout << "After reduction:" << std::endl;
117 //                node->matrix->printMatrix();
118
119                 //4. calculate lowerBound
120                 node->lowerBound = prevNode->matrix->adjMatrix[prevNode->nrV][i] + prevNode->lowerBound +
121                                    reduction; //calculate the current cost
122 //                std::cout << "lowerBound= " << node->lowerBound << std::endl;
```

Rysunek 3-2 Obliczanie wartości dolnego ograniczenia w trakcie algorytmu

Najpierw wypełniony zostaje określony rząd i kolumna wartościami „nieskończoność” (-1), w zależności od wierzchołka w którym aktualnie się znajdujemy oraz z którego wierzchołka osiągnięto aktualny wierzchołek. Następnie należy zapobiec cyklom, tzn. wstawić wartość „nieskończoność” w komórkę o indeksie: [nr_aktualnego_wierzchołka][0]. Kolejnym krokiem jest wywołanie opisanej wcześniej metody *reduceMatrix()*. Wartość dolnego ograniczenia danego wierzchołka to suma:

- Kosztu ścieżki z wierzchołka, z którego osiągnięto aktualny wierzchołek do aktualnego wierzchołka, pobranego z macierzy przechowywanej w poprzednim wierzchołku
- Dolnego ograniczenia wierzchołka, z którego osiągnięto aktualny wierzchołek
- Zwróconej wartości przez wywołaną wcześniej metodę *reduceMatrix()*

- Kolejne kroki algorytmu:

1. Stworzenie nowego wskaźnika na obiekt typu wierzchołek i przypisanie mu parametrów
2. Redukcja macierzy i obliczenie dolnego ograniczenia
3. Stworzenie kolejki priorytetowej i dodanie do niej wierzchołka początkowego
4. Dopóki kolejka nie jest pusta, powtarzaj kroki 4-20:
5. Sprawdź czy dolne ograniczenie wierzchołka znajdującego się na początku kolejki jest mniejsze od górnego ograniczenia
    a. Jeśli NIE: została znaleziona najbardziej optymalna ścieżka, zakończ algorytm
6. Usuń wierzchołek z kolejki
7. Dopóki kolejka nie jest pusta, powtarzaj kroki 8-10:
8. Sprawdź czy w kolejce znajdują się jeszcze jakieś wierzchołki o dolnym ograniczeniu mniejszym od górnego ograniczenia
    a. Jeśli tak, przenieś je do kolejki priorytetowej pomocniczej
    b. Jeśli nie, usuń je
9. Skopiuj kolejkę pomocniczą do kolejki głównej
10. Usuń wierzchołki z kolejki pomocniczej
11. Dla każdego sąsiada wierzchołka wyciągniętego z kolejki: wykonuj kroki 11-19
12. Sprawdź czy wierzchołek został już odwiedzony
    a. Jeśli tak, to następny obieg pętli z kroku 10.
13. Stwórz nowy wierzchołek o parametrach skopiowanych z wierzchołka poprzedniego
14. Dodaj numer nowo stworzonego wierzchołka do drogi i zwiększ poziom o 1
15. Zmniejsz rozmiar macierzy
16. Zapobiegnij cyklom
17. Zredukuj macierz
18. Oblicz dolne ograniczenie
19. Wstaw wierzchołek do kolejki
20. Sprawdź czy dolne ograniczenie wierzchołka jest mniejsze od górnego ograniczenia
21. Jeśli odwiedzono wszystkie wierzchołki, zapisz ciąg odwiedzonych wierzchołków najlepszego rozwiązania oraz zaktualizuj wartość górnego ograniczenia
22. ZAKOŃCZ

Przykład:

Załóżmy, że dany jest graf o następującej reprezentacji macierzowej:

$$\begin{bmatrix} -1 & 9 & 7 \\ 3 & -1 & 0 \\ 0 & 6 & -1 \end{bmatrix}$$

1. Stwórz nowy wierzchołek *prevNode*, skopiuj do niego powyższą macierz i przypisz:

$$prevNode \rightarrow nrV = 0$$
$$prevNode \rightarrow level = 0$$

$$prevNode \rightarrow route = \{0\}$$

2. Zredukuj macierz
   Wiersze:

$$\begin{bmatrix} -1 & 9 & 7 \\ 3 & -1 & 0 \\ 0 & 6 & -1 \end{bmatrix} \begin{matrix} 7 \\ 0 \\ 0 \end{matrix}$$

$$\begin{bmatrix} -1 & 2 & 0 \\ 3 & -1 & 0 \\ 0 & 6 & -1 \end{bmatrix}$$

Kolumny:

$$\begin{bmatrix} -1 & 2 & 0 \\ 3 & -1 & 0 \\ 0 & 6 & -1 \end{bmatrix}$$
$$\begin{matrix} 0 & 2 & 0 \end{matrix}$$

$$prevNode \rightarrow matrix = \begin{bmatrix} -1 & 0 & 0 \\ 3 & -1 & 0 \\ 0 & 4 & -1 \end{bmatrix}$$
$$lowerBound = 7 + 0 + 0 + 0 + 2 + 0 = 9$$

3. Dodaj wierzchołek początkowy do kolejki

$$priorityQueue = \{< 9; * prevNode >\}$$

4. Dopóki kolejka nie jest pusta:

$$9 < MAX_{INT}$$

Więc usuń wierzchołek z kolejki:

$$priorityQueue = \emptyset$$

Kolejka jest pusta.
i=0, wierzchołek odwiedzony
i=1, nieodwiedzony, więc:
Stwórz nowy wierzchołek i skopiuj macierz z wierzchołka usuniętego z kolejki

$$node \rightarrow nrV = 1$$
$$node \rightarrow level = 0 + 1 = 1$$
$$node \rightarrow route = \{0,1\}$$

Zmniejsz rozmiar macierzy:

$$\begin{bmatrix} -1 & -1 & -1 \\ 3 & -1 & 0 \\ 0 & -1 & -1 \end{bmatrix}$$

Zapobiegnij cyklom:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & -1 & 0 \\ 0 & -1 & -1 \end{bmatrix}$$

Redukcja:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & -1 & 0 \\ 0 & -1 & -1 \end{bmatrix} \begin{matrix} 0 \\ 0 \\ 0 \end{matrix}$$

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & -1 & 0 \\ 0 & -1 & -1 \end{bmatrix}$$
$$0 \quad 0 \quad 0$$

$$reduction = 0$$
$$node \rightarrow lowerBound = prevNode \rightarrow matrix[0][1] + prevNode \rightarrow lowerBound + reduction = 3 + 9 + 0 = 12$$

Wstaw wierzchołek do kolejki:

$$priorityQueue = \{< 12; * node >\}$$

$$12 < MAX_{INT}$$

Więc:

$$bestSol = node$$
$$minCost = 12$$

i=2 nieodwiedzony, więc:
Stwórz nowy wierzchołek i skopiuj macierz z wierzchołka usuniętego z kolejki

$$node \rightarrow nrV = 2$$
$$node \rightarrow level = 0 + 1 = 1$$
$$node \rightarrow route = \{0,2\}$$

Zmniejsz rozmiar macierzy:

$$\begin{bmatrix} -1 & -1 & -1 \\ 3 & -1 & -1 \\ 0 & 4 & -1 \end{bmatrix}$$

Zapobiegnij cyklom:

$$\begin{bmatrix} -1 & -1 & -1 \\ 3 & -1 & -1 \\ -1 & 4 & -1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & -1 & -1 \\ 3 & -1 & -1 \\ -1 & 4 & -1 \end{bmatrix}$$

Redukcja:

$$\begin{bmatrix} -1 & -1 & -1 \\ 3 & -1 & -1 \\ -1 & 4 & -1 \end{bmatrix} \begin{matrix} 0 \\ 3 \\ 4 \end{matrix}$$

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & -1 & -1 \\ -1 & 0 & -1 \end{bmatrix}$$
$$0 \quad 0 \quad 0$$

$$reduction = 7$$

$$node \rightarrow lowerBound = prevNode \rightarrow matrix[0][2] + prevNode \rightarrow lowerBound + reduction = 0 + 9 + 7 = 16$$

Wstaw wierzchołek do kolejki:

$$priorityQueue = \{< 12; *node >, < 16, *node >\}$$

$$16 > 12$$

Więc nie jest to wierzchołek o najmniejszym dolnym ograniczeniu.

$$bestSol \rightarrow level = 1\ ! = 2$$

Więc nie podmieniamy rozwiązania.

Kolejny obieg pętli:

$$12 < MAX_{INT}$$

Więc usuń wierzchołek z kolejki:

$$priorityQueue = \{< 16, *node >\}$$

Jeśli w kolejce pozostały wierzchołki, w których lowerBound > upperBound, usuń je

$$16 < MAX_{INT}$$

Więc pozostawiamy wierzchołek w kolejce.

i=0 wierzchołek odwiedzony
i=1 wierzchołek odwiedzony
i=2 wierzchołek nieodwiedzony, więc:

Stwórz nowy wierzchołek i skopiuj macierz z wierzchołka usuniętego z kolejki

$$node \rightarrow matrix = \begin{bmatrix} -1 & -1 & -1 \\ -1 & -1 & 0 \\ 0 & -1 & -1 \end{bmatrix}$$

$$node \rightarrow nrV = 2$$
$$node \rightarrow level = 1 + 1 = 2$$
$$node \rightarrow route = \{0,1,2\}$$

Zmniejszenie rozmiaru macierzy:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & -1 & -1 \\ 0 & -1 & -1 \end{bmatrix}$$

Zapobiegnij cyklom:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & -1 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Zredukuj macierz:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & -1 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Nic nie zostanie zredukowane.

$$reduction = 0$$
$$lowerBound = prevNode \rightarrow matrix[1][2] + prevNode \rightarrow lowerBound + reduction$$
$$= 0 + 12 + 0 = 12$$

Wstaw wierzchołek do kolejki:

$$priorityQueue = \{< 12; * node >, < 16, * node >\}$$

$$12 < MAX_{INT}$$

Więc:

$$bestSol = node$$
$$minCost = 12$$

$$bestSol \rightarrow level = 2$$

Więc to jest nowe najlepsze rozwiązanie:

$$solution = \{0,1,2\}$$
$$minCost = 12$$

Kolejny obieg pętli:

$$12 = 12$$

Więc zakończ algorytm.

Wynik:
Ciąg wierzchołków: 0->1->2
Koszt: 12

4. Opis implementacji algorytmów

W przypadku obu algorytmów graf jest przechowywany w pamięci komputera w postaci macierzy ($N\ x\ N$).

- Brute – force

  Do wygenerowania kolejnych permutacji wykorzystywana jest jednowymiarowa tablica dynamiczna o rozmiarze N.

- Branch & Bound

  W implementacji tego algorytmu wykorzystano kolejkę priorytetową z biblioteki STL. Każdy element kolejki to para: <klucz, wartość>, gdzie jako klucz przyjmuje się wartość dolnego ograniczenia (lowerBound) a jako wartość – wskaźnik na obiekt typu Node, który ma następujące pola:
  - Macierz

- Wektor typu int – przechowuje ścieżkę, którą osiągnięto dany wierzchołek (włączając w to aktualny numer wierzchołka)
- Zmienna lowerBound – przechowuje dolne ograniczenie danego wierzchołka
- Zmienna nrV – przechowuje numer wierzchołka
- Zmienna level – przechowuje ilość odwiedzonych wierzchołków nie licząc wierzchołka początkowego

Dodatkowo kolejkę priorytetową, która jest domyślnie implementacją kopca maksymalnego, zmodyfikowano na implementację kopca minimalnego poprzez podanie do jej parametrów przy jej deklaracji klasy *Cmp.h* zawierającą metodę *operator()*. Kolejkę priorytetową zastosowano w celu łatwego dostępu do wierzchołka o najmniejszej wartości dolnego ograniczenia.

5. Plan eksperymentu

Reprezentacja macierzowa grafu jest generowana w klasie *Randomizer*. Na początku macierz wypełniana jest wartością „-1”. Następnie dla każdej komórki w macierzy, z wyjątkiem tych o tych samych indeksach i oraz j, jest generowana liczba całkowita z przedziału [1;100].

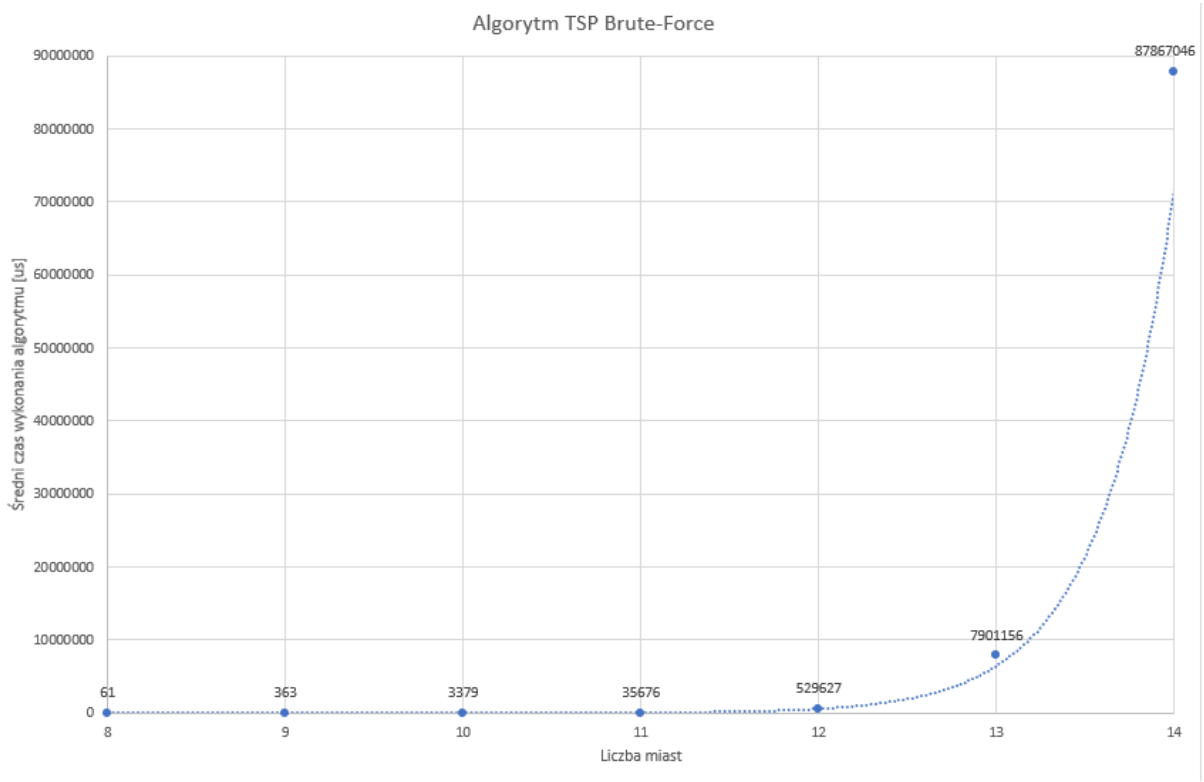
Czas mierzony jest w mikrosekundach za pomocą klasy *Timer*, za pomocą biblioteki <*windows.h*>. Po wywołaniu metody startTimer() zostaje odczytana zostaje liczba „tiknięć” od ostatniego restartu systemu. Następnie po wywołaniu metody stopTimer(), znowu odczytana zostaje liczba „tiknięć” a następnie zostaje ona odjęta od wartości odczytanej z wywołaniem metody startTimer() i podzielona przez wartość *frequency = 10MHz*.

Na czas testów tworzona jest 100 elementowa tablica, do której wpisywany jest czas wykonania dla każdej instancji. Na koniec testów, wszystkie czasy wypisywane są do pliku z rozszerzeniem .txt, w celu kontrolnym, czy nie następują jakieś rozbieżności. Następnie liczona jest średnia ze wszystkich czasów, która jest także wypisywana do pliku .txt . Nazwa pliku .txt ma następujący format: „typAlgorytmu_liczbaMiast.txt”.

Wykonano eksperymenty dla 7 reprezentatywnych wartości N dla każdego algorytmu. W przypadku algorytmu Branch & Bound ustalono 2 minutowy limit czasowy. Jeśli algorytm nie został wykonany w ustalonym limicie czasu, przerywano go i nie brano jego długości wykonania pod uwagę w trakcie liczenia średniego czasu wykonania algorytmu.

- Brute – force

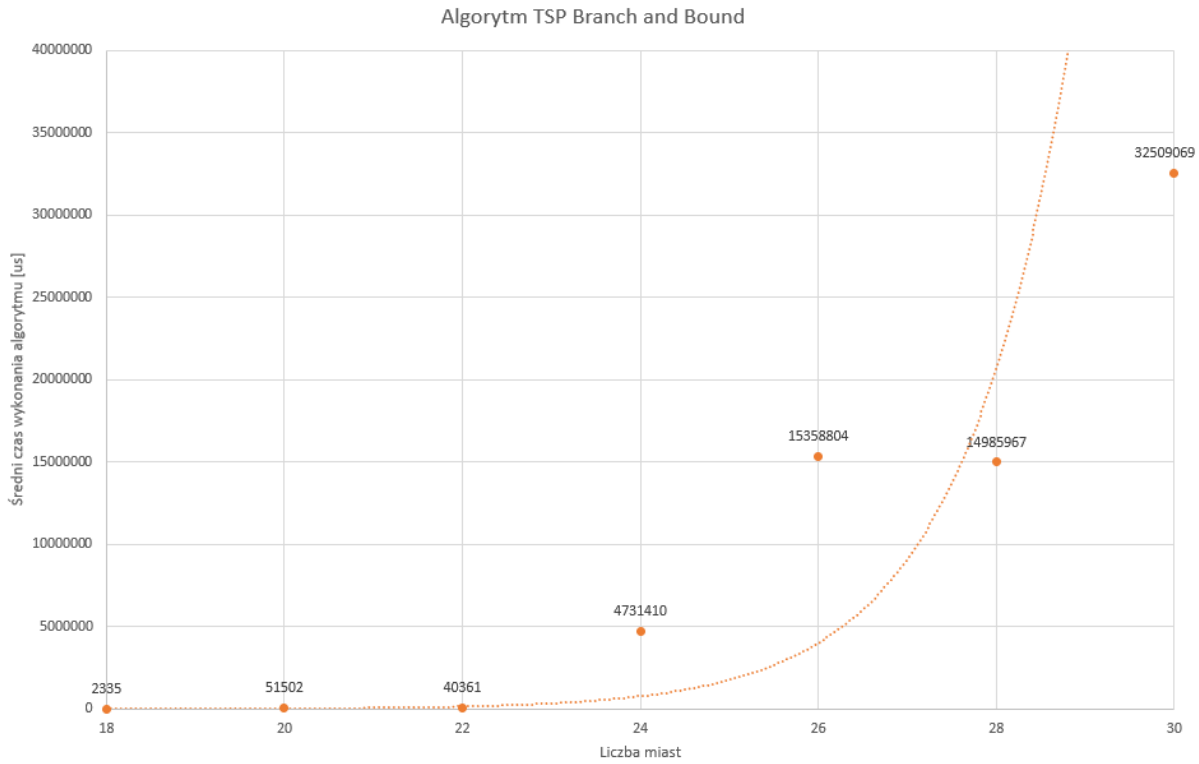
L.P.	N	Średni czas wykonania algorytmu [us]
1	8	61
2	9	363
3	10	3379
4	11	35676
5	12	529627
6	13	7901156
7	14	87867046



Rysunek 5-1 Wykres przedstawiający czas wykonania algorytmu TSP brute-force

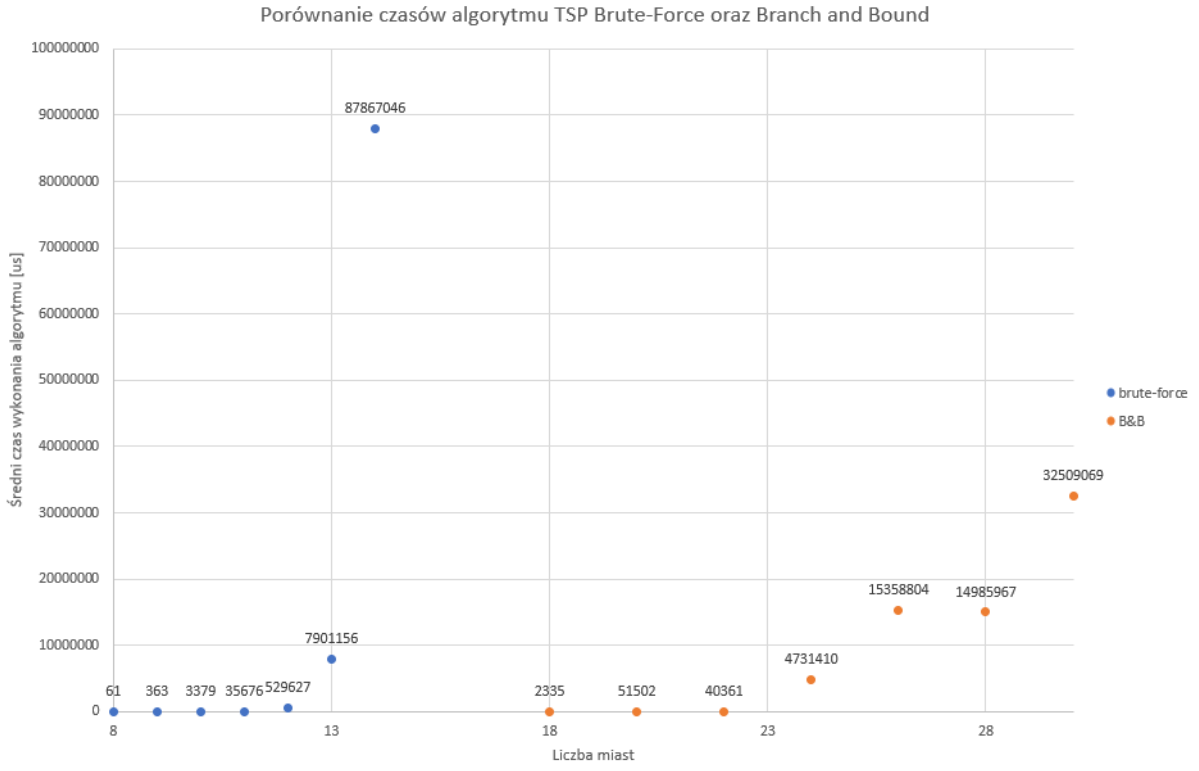
- Branch & Bound

L.P.	N	Średni czas wykonania algorytmu [us]	Pominięte instancje [%]
1	18	2335	0%
2	20	51502	0%
3	22	40361	0%
4	24	4731410	0%
5	26	15358804	9%
6	28	14985967	38%
7	30	32509069	57%



Rysunek 5-2 Wykres przedstawiający czas wykonania algorytmu TSP B&B

Porównanie obu algorytmów:



Rysunek 5-3 Porównanie czasów wykonania algorytmu TSP brute-force oraz B&B

6. Wnioski

Algorytm brute-force okazał się trwać znacznie dłużej niż 5 minut dla liczby miast powyżej 14. Branch and bound natomiast do liczby 24 miast dla każdej ze 100 losowych instancji był w stanie wygenerować wynik poniżej 2 minut. Zastosowanie ograniczenia w algorytmie B&B pozwoliło na usprawnienie algorytmu komiwojażera. Przeprowadzone eksperymenty potwierdziły teoretyczne złożoności obliczeniowe obu algorytmów.

7. Źródła

- https://www.youtube.com/watch?v=1FEP_sNb62k&ab_channel=AbdulBari
-