Implementacja i analiza efektywności algorytmów BF i B&B.

Zadanie projektowe nr 1

Prowadzący: dr inż. Jarosław Mierzwa

Grupa: Piątek 15.15-16.55

Autor: Izabela Pałubicka 252771

1. Opis asymetrycznego problemu komiwojażera.

1.1.Problem komiwojażera

Problem komiwojażera (ang. The Travelling Salesman Problem (TSP)) polega na znalezieniu minimalnego cyklu Hamiltona w grafie pełnym ważonym. W takim cyklu każdy wierzchołek grafu musi zostać odwiedzony dokładnie raz, a sam cykl kończy się powrotem do wierzchołka startowego.

Istnieją dwie wersje tego problemu:

- symetryczna, gdzie odległość od miasta *i* do miasta *j* jest taka sama jak odległość z miasta *j* do miasta *i*
- asymetryczna, gdzie odległości między miastami mogą być różne.

1.2. Metoda przeglądu zupełnego

Metoda przeglądu zupełnego (ang. Brute Force) jest to metoda, która generuje wszystkie możliwe rozwiązania, sprawdza je wyliczając funkcje celu i wybiera optymalne rozwiązanie. Jest to metoda dokładana, która daję gwarancję znalezienia najlepszego wyniku. Częstą zaletą tego narzędzia jest łatwa implementacja algorytmu opartego na tej metodzie. Wadą jest duża złożoność obliczeniowa.

W problemie komiwojażera znajduje swe zastosowanie przez wygenerowanie wszystkich możliwych tras i wyborze tej, której koszt, czyli suma wartości wszystkich dróg między kolejnymi miastami, jest najmniejszy.

Złożoność obliczeniowa dla problemu komiwojażera rozwiązanego tą metodą wynosi: **n!.**

1.3. Metoda podziału i ograniczeń

Metoda podziału i ograniczeń (ang. Branch and Bound) jest to metoda, która polega na analizowaniu drzewa przestrzeni stanów. Drzewo to reprezentuje wszystkie możliwe drogi jakimi może pójść algorytm podczas rozwiązywania problemu. Przeszukiwanie całego drzewa byłoby kosztowne (pamięciowo i czasowo), dlatego dla każdego węzła obliczane jest ograniczenie, co pozwala na stwierdzenie czy dane rozwiązanie ma szanse być optymalne. Jeśli dany węzeł nie ma szansy być rozwiązaniem optymalnym, nie przeszukuje się jego potomków. Takie narzędzie pozwala na zmniejszenie ilości odwiedzanych wierzchołków i przyspieszyć odnalezienie rozwiązania. Algorytm korzystający z metody Branch and

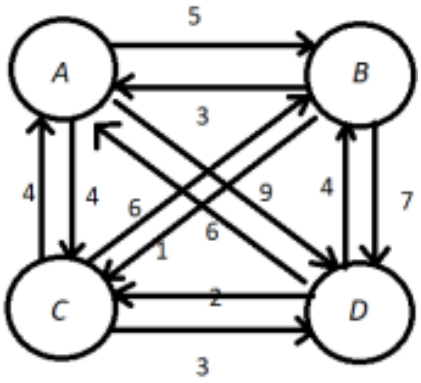
Bound musi posiadać dwa elementy, które zależne są od rodzaju problemu: funkcję obliczającą ograniczenie i strategię odwiedzin wierzchołków.

W projekcie do odwiedzin wierzchołków została użyta strategia Best First

Złożoność obliczeniowa dla problemu komiwojażera rozwiązanego tą metodą może wynieść w najlepszym przypadku **n**, a w najgorszym tyle samo co dla przeszukiwania zupełnego czyli **n!.**

2. Opis działania algorytmu Branch and Bound krok po kroku.

Działanie algorytmu zostanie przedstawione za pomocą poniższego przykładu:



Wypisanie początkowej macierzy kosztów:

$$\begin{bmatrix} -1 & 5 & 4 & 9 \\ 3 & -1 & 1 & 8 \\ 4 & 6 & -1 & 3 \\ 6 & 4 & 2 & -1 \end{bmatrix}$$

Zamiana -1 na ∞:

$$\begin{bmatrix} \infty & 5 & 4 & 9 \\ 3 & \infty & 1 & 8 \\ 4 & 6 & \infty & 3 \\ 6 & 4 & 2 & \infty \end{bmatrix}$$

1)Tworzymy korzeń drzewa, którego wierzchołki odpowiadają miastom. Korzeniem będzie wierzchołek A, z którego zaczynamy drogę.

Aby wyliczyć dolne ograniczenie korzenia należy dokonać redukcji macierzy. Polega to na redukcji wierszy i kolumn macierzy, czyli na wybraniu najmniejszej wartości w wierszu/kolumnie i odjęciu tej wartości od pozostałych

z danego wiersza/danej kolumny. Gdy w danym wierszu/kolumnie występują same nieskończoności lub chociaż jedno zero, wiersz/kolumnę uznajemy za zredukowane.

$$\left[\begin{array}{cccc|c} \infty & 5 & 4 & 9 & 4 \\ 3 & \infty & 1 & 8 & 1 \\ 4 & 6 & \infty & 3 & 3 \\ 6 & 4 & 2 & \infty & 2 \end{array}\right] \Rightarrow \left[\begin{array}{cccc} \infty & 1 & 0 & 5 \\ 2 & \infty & 0 & 7 \\ 1 & 3 & \infty & 0 \\ 4 & 2 & 0 & \infty \\ \hline 1 & 1 & x & x \end{array}\right] \Rightarrow \left[\begin{array}{cccc} \infty & 0 & 0 & 5 \\ 1 & \infty & 0 & 7 \\ 0 & 2 & \infty & 0 \\ 3 & 1 & 0 & \infty \end{array}\right]$$

Ograniczenie minimalne (lower bound) = 4 + 1 + 3 + 2 + 1 + 1 = 12

Znając ograniczenie minimalne mamy pewność, że rozwiązanie nie będzie mniejsze od danego ograniczenia.

Następnie rozważamy wszystkie pozostałe wierzchołki, do których możemy się dostać z aktualnie badanego węzła. Wybieramy najlepszy wierzchołek, aby zminimalizować koszt trasy.

Obliczanie ograniczenia dla poziomów n > 0 badanego drzewa wygląda podobnie. Najpierw należy zablokować odpowiedni wiersz, kolumnę i powrót do wierzchołka startowego, a następnie można przejść do redukcji w taki sam sposób jak dla korzenia.

Ogólny wzór na obliczenie dolnego ograniczenia :

$$Lower\ bound\ =\ cost(parent)\ +\ redukcja\ +\ M[A,B]$$

Gdzie
cost(parent) – dolne ograniczenie ojca
redukcja – koszt redukcji macierzy
M[A, B] – koszt przejścia z ojca do dziecka (badanego węzła) pobrany z macierzy ojca

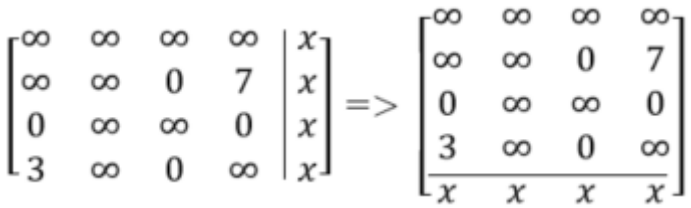
2) Badamy wierzchołek B; (droga A->B)

cost(parent) = 12

M[A, B] = 0

Liczymy redukcję dla wierzchołka B.

Kopiujemy tablicę ojca i blokujemy kolumnę B, wiersz A i powrót do ojca M[B,A] przez znak ∞, a następnie przechodzimy do redukcji.



redukacja = 0

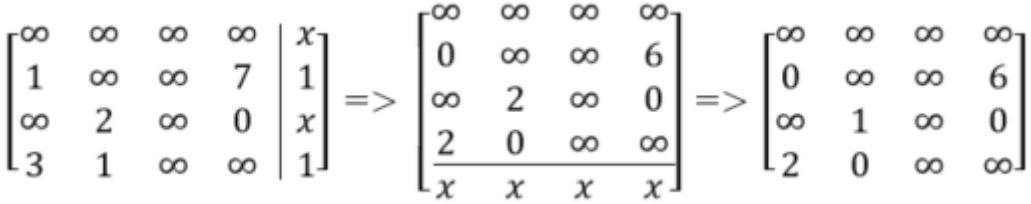
lower bound = 12 + 0 + 0 = **12**

3) Badamy wierzchołek C (droga A->C)

cost(parent) = 12

M[A, C] = 0

Kopiowanie macierzy, blokowanie i redukcja:



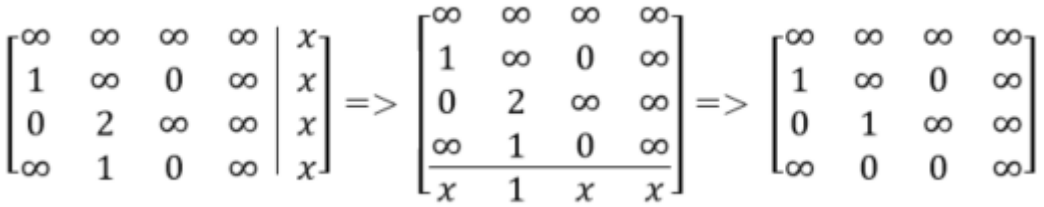
lower bound = 12 + 2 + 0 = 14

4) Badamy wierzchołek D (droga A->D)

cost(parent) = 12

M[A, D] = 5

Kopiowanie macierzy, blokowanie i redukcja:



lower bound = 12 + 1 + 5 = **18**

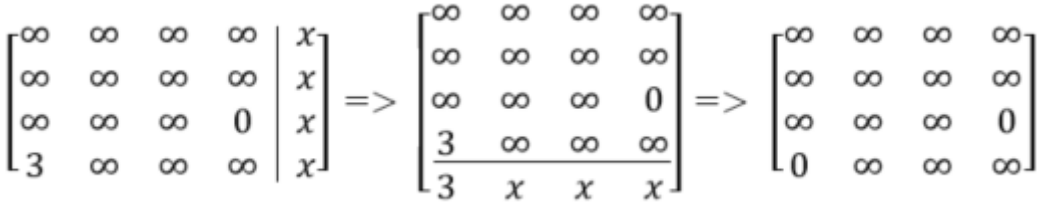
Dolne ograniczenie wyszło najmniejsze dla punktu B, dlatego dalej rozwijana jest ta ścieżka.

5) Badamy wierzchołek C; (droga A->B->C)

cost(parent) = 12

M[B, C] = 0

Kopiowanie macierzy, blokowanie i redukcja:



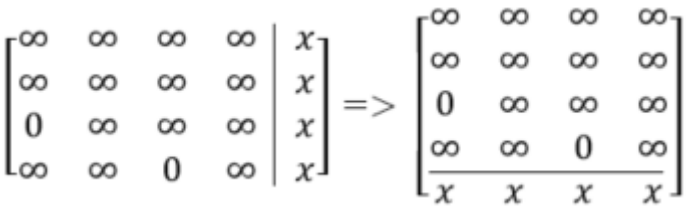
Lower bound = 12 + 3 + 0 = 15

6) Badamy wierzchołek D; (droga A->B->D)

cost(parent) = 12

M[B, D] = 7

Kopiowanie macierzy, blokowanie i redukcja:



Lower bound = 12 + 0 + 7 = 19

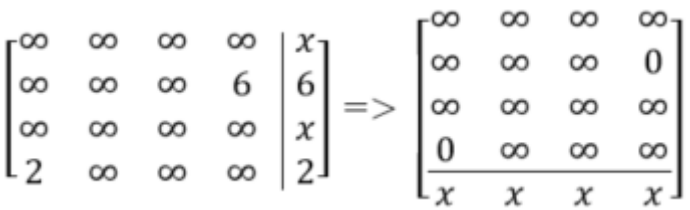
Droga A->C ma mniejszy koszt (14) niż koszt A->B->C (15), dlatego wracamy do węzła C, którego ojcem jest A.

7) Badamy wierzchołek B; (droga A->C->B)

cost(parent) = 14

M[C, B] = 0

Kopiowanie macierzy, blokowanie i redukcja:



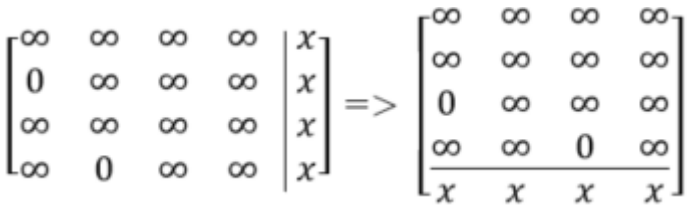
Lower bound = 14 + 8 + 1 = **23**

8) Badamy wierzchołek D; (droga A->C->D)

cost(parent) = 14

M[C, D] = 0

Kopiowanie macierzy, blokowanie i redukcja:



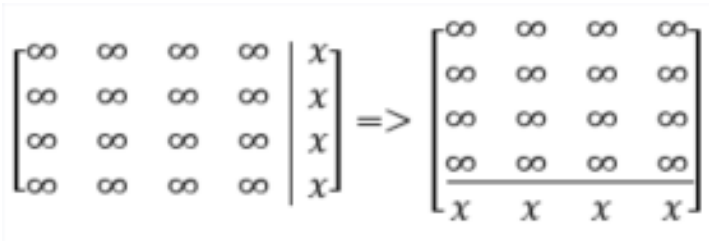
Lower bound = 14 + 0 + 0 = **14**

9) Badamy wierzchołek B; (droga A->C->D->B)

cost(parent) = 14

M[D,B] = 0

Kopiowanie macierzy, blokowanie i redukcja:



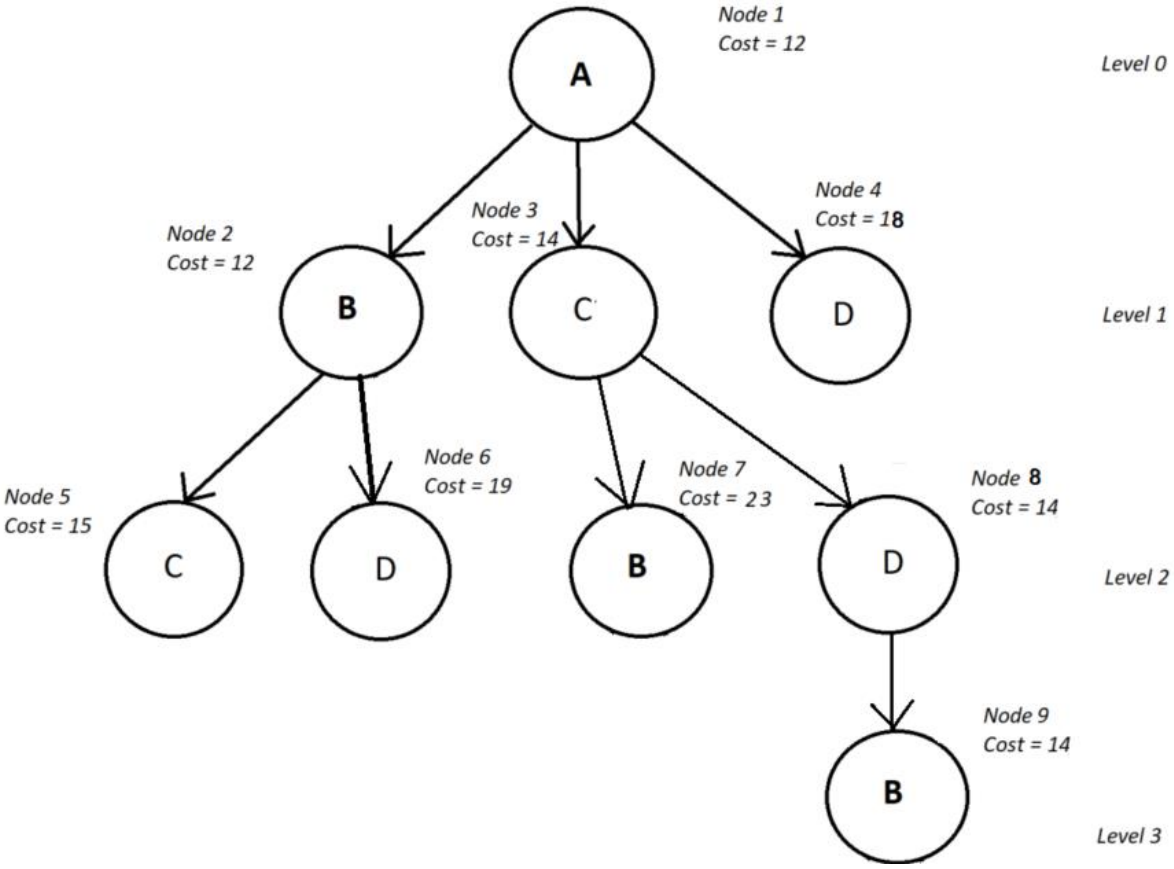
Lower bound = 14 + 0 + 0 = **14**

Rozwiązanie:

Ścieżka: **A->C->D->B->A**

Koszt drogi: **14**

Drzewo przeszukiwań dla powyższego zadania:



3. Opis implementacji algorytmu Branch and Bound

3.1. Wykorzystane struktury

- Graf pełny alokowany jest w dynamicznej tablicy dwuwymiarowej
- Węzły drzewa przeszukiwań alokowane dynamicznie posiadają informacje o:
  - numerze węzła
  - ścieżce, czyli jak dotarto do węzła od wierzchołka startowego (jest to wektor krawędzi, które zawierają informacje o węzłach z i do)
  - koszcie danego węzła (lower bound)
  - tablicy, kopiowanej od ojca, a następnie redukowanej do przeliczenia kosztu węzła i aby mogła zostać skopiowana do następnego węzła
  - poziom drzewa, który informuje i ilości wierzchołków odwiedzonych
- Kolejka priorytetowa, która porządkuje węzły od najmniejszego kosztu

3.2. Funkcja obliczająca ograniczenia

Ograniczenie dla węzła jest liczone w następujący sposób:

```
// Koszt = koszt ojca + koszt przejści z rodzica do dziecka + redukcja
child->cost = minimalNode->cost + minimalNode->reducedMatrix[minimalNode->vertex][i]
    + calculateReduction(child->reducedMatrix);
```

Koszt dziecka liczony jest na podstawie kosztu ojca, drogi od ojca do dziecka przechowywanej w tablicy ojca i funkcji obliczającej redukcję macierzy.

Funkcja redukcji:

```
int BranchAndBound::calculateReduction(int** reducedMatrix)
{
    int cost = 0;

    int* row = new int[numberOfNodes];
    int* column = new int[numberOfNodes];

    // Inicjalizowanie wszystkich wartości tablic jako INT_MAX
    fillWithInfinities(row);
    fillWithInfinities(column);

    // Redukowanie macierzy kosztów
    rowReduction(reducedMatrix, row);
    columnReduction(reducedMatrix, column);

    // Wartość redukcji jest sumą wszystkich redukcji
    for (int i = 0; i < numberOfNodes; i++)
    {
        if (row[i] != INT_MAX)
            cost += row[i];
        if (column[i] != INT_MAX)
            cost += column[i];
    }
    delete[] column;
    delete[] row;
    return cost;
}
```

Alokowane są dynamicznie dwie tabele do przechowywania minimalnych wartości z wierszy/kolumn i wypełniane są początkowo wartościami INT_MAX funkcją fillWithInfinities.

Następnie macierz jest redukowana wierszami funkcją rowReduction, w której z każdego wiersza wyszukiwany jest najmniejszy koszt i zapisywany jest w tabeli row. Następnie wartości te są odejmowane od wartości z wierszy, o ile wartość w row lub wartość z wiersza nie jest równa INT_MAX.

Analogicznie dzieje się w funkcji columnReduction.

Następnie dodawane są wszystkie redukcje z wierszy i kolumn z pominięciem INT_MAX.

Koszt korzenia to po prostu redukcja macierzy, ponieważ nie ma ojca i żadna ścieżka na początku do niego nie prowadzi.

4. Plan eksperymentu.

Program został napisany w języku C++ w środowisku Visual Studio i testowany był w wersji Release.

W eksperymencie wartości krawędzi przechowywane były w dynamicznie alokowanej tablicy dwuwymiarowej. Na podstawie podanej ilości węzłów program generował losowo wartość każdej krawędzi z przedziału od 1 do 100, a na przekątnej macierzy wypisał wartości -1. Aby zwiększyć losowość generowania danych użyto funkcji srand(time(0)).

Wykonano po 100 pomiarów czasu (za każdym razem inne koszty krawędzi) i wyliczona została średnia dla instancji o ilości węzłów:

- dla Brute Force: 6, 7, 8, 9, 10, 11, 12;
- dla Branch and Bound: 6, 8, 10, 12, 14, 16, 18;

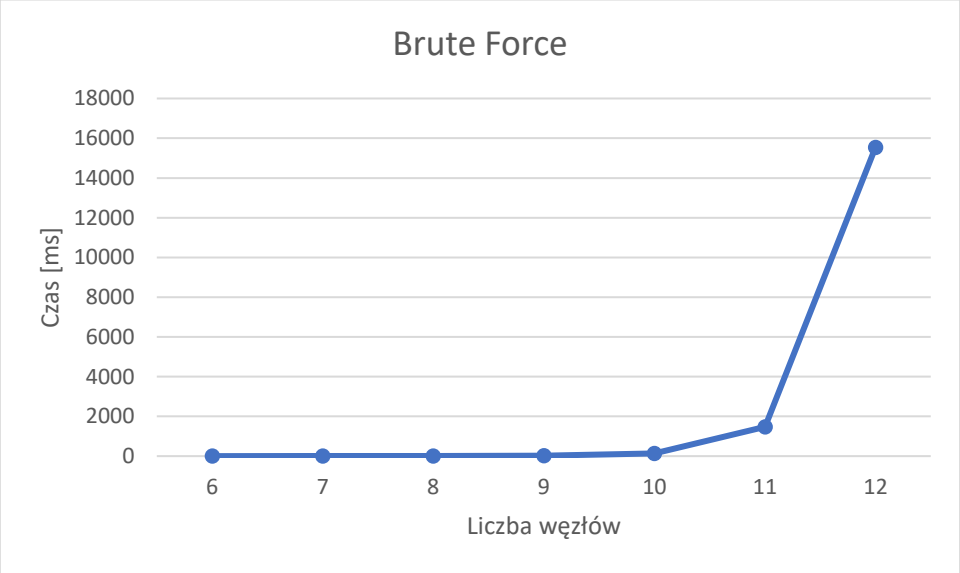
Do pomiaru czasu wykorzystano funkcję std::chrono::high_resolution_clock z biblioteki <chrono>. Pomiary czasów algorytmów nie obejmowały czasu generacji elementów i wyświetlania wyniku. Podczas mierzenia czasu zbędne aplikacje były wyłączone.

5. Wyniki eksperymentu

Wyniki są wyliczoną średnią z poszczególnych pomiarów i podawane są w ms. Przybliżenie dla zwiększenia czytelności to 2 miejsca po przecinku.

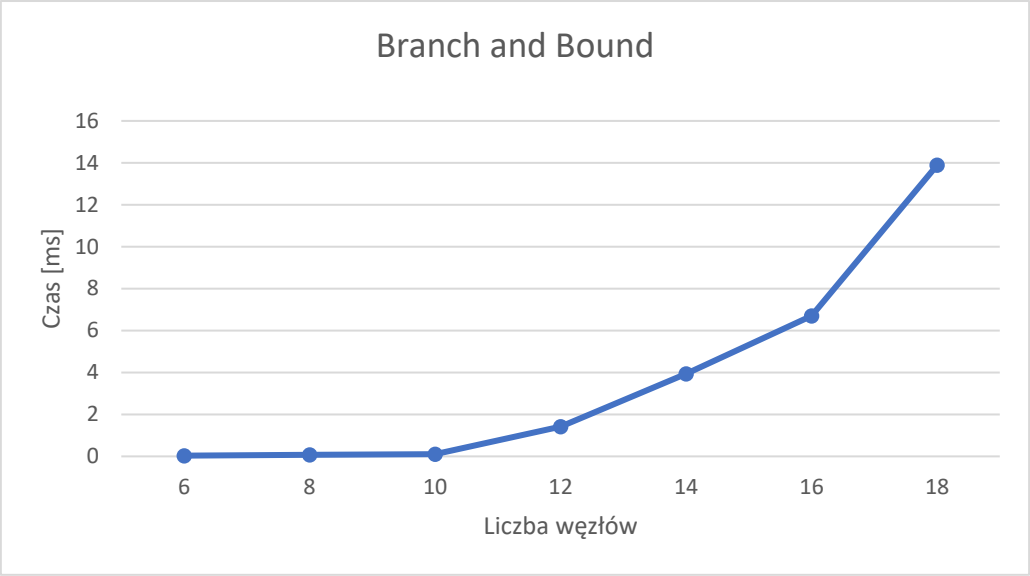
Wyniki dla Brute Force:

Liczba węzłów	Czas [ms]
6	0,05
7	0,50
8	1,82
9	14,90
10	136,22
11	1468,72
12	15542,80

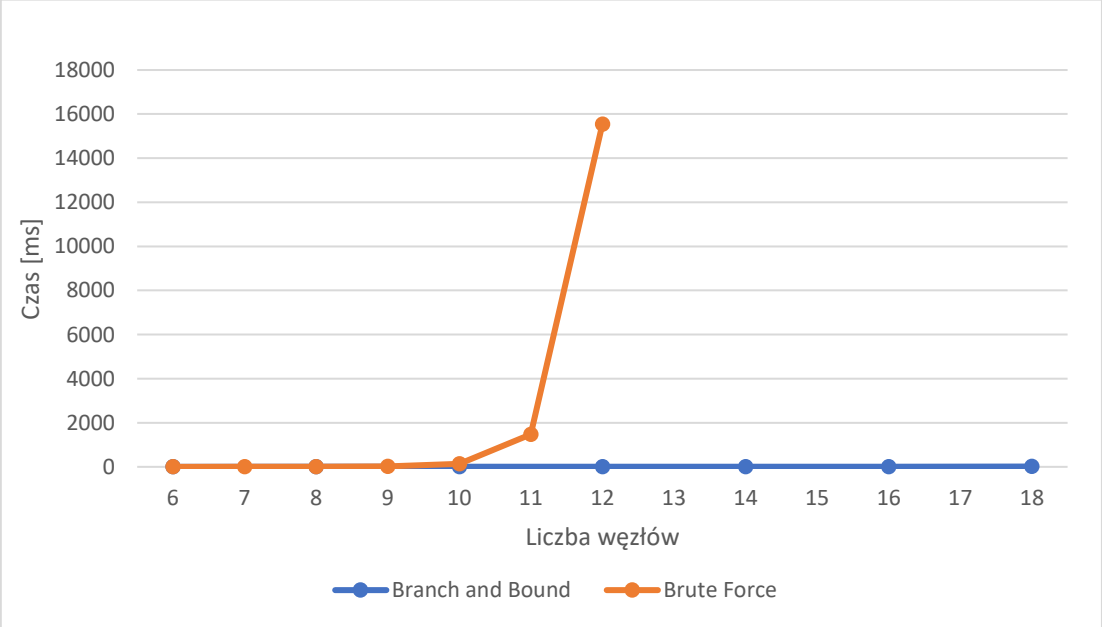


Wyniki dla Branch and Bound:

Liczba węzłów	Czas [ms]
6	0,03
8	0,08
10	0,10
12	1,42
14	3,94
16	6,71
18	13,89



Porównanie wyników dla obu metod:



Maksymalne wielkości instancji dla obu metod:

Metoda	Max ilość węzłów	Opis
Brute Force	13	Dla 14 czas powyżej 5 min
Branch and Bound	34	Dla 35 zabrakło pamięci

6. Wnioski

Metoda Brute Force sprawdza się tyko dla instancji o małej ilości węzłów, ponieważ czas oczekiwania zaczyna gwałtownie rosnąć przy instancjach o wielkości 12/13. Branch and Bound jest zdecydowanie szybszym algorytmem i potrafi zbadać instancje o rozmiarach powyżej 30 w bardzo krótkim czasie. Dzieje się tak, ponieważ przeszukiwane są tylko węzły i ścieżki, które mają potencjał bycia rozwiązaniem. W tym algorytmie czas nie zależy tylko od wielkości instancji, ale też od szczęścia w przeszukiwaniu, ponieważ dla większych instancji potrafił pokazać mniejsze czasy, czego w uśrednionych wynikach nie można było zauważyć. Jest zdecydowanie bardziej efektywny, ale z racji tego, że jest bardzo pamięciochłonny program mógł się wykonać tylko do instancji o 34 węzłach. Oba algorytmy dają pewność znalezienia poprawnego rozwiązania.

Źródła:

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

https://www.gatevidyalay.com/travelling-salesman-problem-using-branch-and-bound-approach/

https://cs.pwr.edu.pl/zielinski/lectures/om/mow10.pdf