# Projektowanie efektywnych algorytmów

Projekt pierwszy

**Implementacja i analiza efektywności algorytmów przeglądu zupełnego , podziału i ograniczeń  oraz programowania dynamicznego**

**Andrzej Olszewski 252737**

**Prowadzący: mg.inż Antoni Sterna**

# 1. Wstęp

## 1.1. Opis zadania

Zadanie polega na zaimplementowaniu oraz dokonaniu analizy efektywności algorytmu przeglądu zupełnego, podziału i ograniczeń (B&B) oraz programowania dynamicznego (DP) dla asymetrycznego problemu komiwojażera (ATSP). Problem komiwojażera jest to zagadnienie optymalizacyjne polegające na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym (pełny graf ważony to taki, w którym wszystkie krawędzi są ze sobą połączone i posiadają etykiety liczbowe przy krawędziach, które wyrażają wagę). Cykl Hamiltona jest to taki cykl w grafie, w którym każdy wierzchołek grafu został odwiedzony dokładnie raz w wyłączeniem pierwszego wierzchołka.

## 1.2. Założenia

Podczas realizacji zadania należy przyjąć następujące założenia:

- używane struktury danych powinny być alokowane dynamicznie (w zależności od aktualnego rozmiaru problemu),
- program powinien umożliwić weryfikację poprawności działania algorytmu (wczytanie danych wejściowych z pliku tekstowego),
- po zaimplementowaniu i sprawdzeniu poprawności działania algorytmu należy dokonać pomiaru czasu jego działania w zależności od rozmiaru problemu N (badania należy wykonać dla minimum 7 różnych reprezentatywnych wartości N),
- dla każdej wartości N należy wygenerować po 100 losowych instancji problemu (w sprawozdaniu należy umieścić tylko wyniki uśrednione),
- implementacje algorytmów powinny być zgodne z obiektowym paradygmatem programowania,
- używanie „okienek” nie jest konieczne i nie wpływa na ocenę (wystarczy wersja konsolowa),
- kod źródłowy powinien być komentowany,
- warto pamiętać o dużych różnicach w wynikach testów czasowych pomiędzy wersjami Debug i Release (testy trzeba przeprowadzić w wersji Release),

# 2. Teoria

Główną trudnością związaną z poszukiwaniem rozwiązania dla problemu komiwojażera jest bardzo duża liczba danych do analizy. Liczba wszystkich możliwych kombinacji krawędzi tworzących cykl Hamiltona w grafie o n wierzchołkach dla przypadku ATSP wynosi (n − 1)!. Rozwiązanie danej instancji tego problemu przy użyciu naiwnego algorytmu wykorzystującego metodę przeglądu zupełnego wymaga wygenerowania wszystkich możliwych cykli (kombinacji) oraz wybrania z nich tego o najmniejszej sumie wag budujących go krawędzi. Prowadzi to do wykładniczej klasy złożoności obliczeniowej O(n!), zatem niemożliwym jest uzyskanie rozwiązania dla dużych n w akceptowalnym czasie.

## 2.1. Brute Force

### 2.1.1. Brute Force Teoria

Metoda przeglądu zupełnego, tzw. przeszukiwanie wyczerpujące (eng. Exhaustive serach) bądź metoda siłowa (eng. Brute force), polega na znalezieniu wszystkich potencjalnych rozwiązań problemu oraz ich analizie w celu wybrania tego, które spełnia warunki zadania. Metodę tę stosuje się do rozwiązywania problemów, dla których znalezienie rozwiązania za pomocą innych dokładnych metod jest niemożliwe lub zbyt trudne. Główną wadą metody Brute Force jak wspomniałem wyżej, jest to, że w przypadku wielu rzeczywistych problemów liczba rozwiązań jest zbyt duża.

### 2.1.2. Brute Force implementacja algorytmu

Brute Force oblicza i porównuje wszystkie możliwe premutacje tras w celu określenia najkrótszej ścieżki. Aby rozwiązać problem TSP przy użyciu tej metody, należy obliczyć całkowitą liczbę tras, a następnie wyznaczyć długości każdej z nich wybierając za każdym razem najkrótszą. Złożoność obliczeniowa algorytmu w przypadku problemu TSP przyjmuje postać wykładniczą O(n!). Przykład działania algorytmu można opisać w oparciu podany powyżej pseudocode. W pętli while obliczane będą kolejne permutacje ścieżki, aż do momentu wyczerpania się możliwych połączeń. W przypadku znalezienia krótszej ścieżki będzie ona zapamiętana.

Do wykonania algorytmu zostały uzyte następujące struktury:

- **Tablice** – jedna dwuwymiarowa przechowująca reprezentację grafu oraz druga jednowymiarowa przechowująca ścieżkę

### 2.1.3. Przykład

| | | | |
|---|---|---|---|
| 0 | 10 | 15 | 20 |
| 10 | 0 | 35 | 25 |
| 15 | 35 | 0 | 30 |
| 20 | 25 | 30 | 0 |

Dla podanej macierzy generujemy pierwszą permutacje oraz obliczmy jej koszt:

cost_path 0-1-2-3-0 = 10 + 35 + 30 + 20 = 95

Koszt danej scieżki porównujemy z obecnym najlepszym wynikiem (na poczatku jest to maxymalna wartość zmiennych typu int)

```
If(best_result > cost_path ){
        best_result = cost_path
}
```

Algorytm wykonujemy dla kazdej permutacji ściezki

## 2.2. Branch-N-Bound

### 2.2.1. Branch-N-Bound Teoria

Metoda podziału i ograniczeń, opiera się na przeszukiwaniu drzewa reprezentującego przestrzeń rozwiązań problemu metodą w głąb, wszerz, lub najmniejszym kosztem. Własnością algorytmu jest ograniczanie ilości przeglądanych rozwiązań, co w konsekwencji redukuje ilości czasu potrzebnego do wyznaczenia najlepszej wartości. Algortym składa się znastępujących funkcji

- Podział - dzielenie zbioru rozwiązań reprezentowanego przez węzeł drzewa na rozłączne podzbiory, reprezentowane przez następników tego węzła.
- Ograniczanie - pomijanie w przeszukiwaniu tych gałęzi drzewa, o których wiadomo, że nie zawierają optymalnego rozwiązania w swoich liściach

Rozwiazanie dla kazdego podzbioru znajduje się pomiędzy dolnym a górnym ograniczeniem. Górne ogranicznie jest to najlepsze znalezione do tej pory rozwiazanie, a dolne jest jest pewnym oszacowaniem najlepszego, które może zostać odnalezione w podzbiorze reprezentowanym przez następnika bieżącego węzła. Podczas przegladu drzewa odrzucane są te węzły dla których dolne ograniczenie jest wyższe od wartości górnego ograniczenia. Wartość dolnego ograniczenia musi być obliczona tak, żeby żadne, możliwe do uzyskania w danym poddrzewie rozwiązanie nie było lepsze od wartości tego ograniczenia. Może ona zatem odbiegać od rzeczywistej wartości możliwej do uzyskania ale jednocześnie powinna być ona możliwie jak nabliższa istniejącemu rozwiązaniu.

### 2.2.2. Branch-N-Bound implementacja algorytmu z przeszukiwaniem DFS

Algorytm dokonuje przeglądu drzewa rozwiązań w głąb, obliczając dla następników bieżącego węzła ograniczenia dolne. Wybrany węzeł po jego przejściu, nie jest rozważany podczas dalszego przeglądu. Przgląd kolejnych węzłów odbywa rekurencyjnie, modyfikując przy tym bieżącą scieżkę oraz ograniczenie górne. Do implementacji opisywanego algorytmu zostały wykorzystane struktury danych takie jak:

- **Tablice** – dwuwymiarowa, przechowująca reprezentację grafu oraz cztery jednowymiarowe, pierwsza przechowująca informację o odwiedzonych wierzchołkach oraz druga służąca do obliczania dolnych ograniczeń, trzecia zapamiętująca ścieżkę będącą najlepszym znalezionym rozwiązaniem oraz czwarta przechowująca ścieżkę tymczasową dla danego etapu wykonywania algorytmu..
- **Kolejka priorytetowa** – zawierająca wierzchołki, których priorytet wyznacza obliczona dla każdego z nich wartość ograniczenia dolnego.

Obliczanie ograniczenia dolnego odbywa się poprzez wybór z wierszy macierzy krawędzi o jak najniższej wadze. Krawędzie te są zapamiętywane w tablicy której indeks odpowiada wierzchołkowi z którego wychodzi dana krawędź. Suma wag tych krawędzi wyznacza dolne ograniczenie dla wierchołka początkowego. Ograniczenie to jest aktualizowane i przypisywane do każdego z następników bieżącego węzła. Odbywa się to według reguły przedstawionej na następującym przykładzie.

### 2.2.3. Przykład

Wznaczamy najmniesza wartość dla kazdego rzędu

| | **0** | **1** | **2** | **3** | **Min** |
|---|---|---|---|---|---|
| **0** | INF | 10 | 15 | 20 | 10 |
| **1** | 10 | INF | 35 | 25 | 10 |
| **2** | 15 | 35 | INF | 30 | 15 |
| **3** | 20 | 25 | 30 | INF | 20 |

Dolne ograniczenie dla wierzchołka początkowego (kiedy w trakcie przeglądu nie została przebyta jeszcze żadna krawędź) stanowi suma elementów tablicy pomocniczej.

LB = 10 + 10 + 15 + 20 =55

Następnym krokiem jest obliczenie dolnych ograniczeń dla następników. W tym celu od przekazanej do następnika wartości dolnego ograniczenia poprzednika należy odjąć wartość wagi odpowiedniej krawędzi z tablicy pomocniczej oraz dodać wartość wagi krawędzi przebytej do następnika. Stąd dla kazdego nastepnika korzenia:

$LB_{01}$ = 55 – min[1] + macierz[0][1] = 55

$LB_{02}$ = 55 – min[1] + macierz[0][2] = 60

$LB_{03}$ = 55 – min[1] + macierz[0][3] = 65

Nastepnie wybieramy najmnieszą wartość która jest mniejsza od górnego ograniczenia (do momentu znaleźenia pierwszego rozwiązania jest ono równe nieskonczoność) i powtarzamy algorytm. W momencie gdy dojdziemy do liścia i dolne ograniczenie jest mniejsze od górnego aktualizujemy wartość górnego ogranicznia. Nasępnie pobieramy kolejny wierzchołek która

spełnia warunki algorytmu. Algorytm kończy się gdy nie ma już wierzchołków spełniających warunek że jego dolna granica jest mniejsza od górnej.

### 2.2.4. Branch-N-Bound implementacja algorytmu z przeszukiwaniem low cost

Algorytm ma jedną globalną kolejke priorytetową która zwraca zawsze wierzchołek z najniższym kosztem. Przegląd wezłów odbywa się zgodnie z kolejnością wyznaczoną przez kolejkę priorytetową. Ograniczenie dolne odbywa się poprzez wyznaczenie najmniejszych warości dla każdego rzedu oraz każdej kolumny oraz redukcję macierzy o te koszty tak, że w każdym rzędzie i każdej kolumnie musi być przynajmniej jedno zero. Wartością dolnego ograniczenia jest suma najmnieszych wartości z każdego wiersza oraz każdej kolumny. Struktury wykorzystane przy implementacji:

- Kolejka priorytetowa - zawierająca wierzchołki, których priorytet wyznacza obliczona dla każdego z nich wartość ograniczenia dolnego.
- Tablice - dwuwymiarowa, przechowująca zredukowaną macierz sąsiedztwa, oraz jednowymiarowa zawierająca najlepsza sieżkę

### 2.2.5. Przykład

Zakładamy że korzeniem jest wierzchołek 0

| | **0** | **1** | **2** | **3** |
|---|---|---|---|---|
| **0** | INF | 10 | 15 | 20 |
| **1** | 10 | INF | 35 | 25 |
| **2** | 15 | 35 | INF | 30 |
| **3** | 20 | 25 | 30 | INF |

Szukamy mnimum dla każdego rzędu. Powstaje następująca tablica min_row = [10,10,15,20]. Nastepnie redukujemy każdy rząd.

| | **0** | **1** | **2** | **3** |
|---|---|---|---|---|
| **0** | INF | 0 | 5 | 10 |
| **1** | 0 | INF | 25 | 15 |
| **2** | 0 | 20 | INF | 15 |
| **3** | 0 | 5 | 10 | INF |

Szukamy mnimum dla każdej kolumny. Powstaje następująca tablica min_col =[0,0,5,10]. Nastepnie redukujemy każdy rząd.

| | **0** | **1** | **2** | **3** |
|---|---|---|---|---|
| **0** | INF | 0 | 0 | 0 |
| **1** | 0 | INF | 20 | 5 |
| **2** | 0 | 20 | INF | 5 |
| **3** | 0 | 5 | 5 | INF |

Koszt dla korzenia jest równe sumie min_row i min_col

LB = min_col + min_row = 70

Następnym krokiem jest obliczenie dolnych ograniczeń dla następników.

Dla połączenia (0,1) dla wszystkich wartości w wierszu zerowym i kolumnie pierwszej oraz dla punktu (1,0) ustawiamy nieskończoność.

| | **0** | **1** | **2** | **3** |
|---|---|---|---|---|
| **0** | INF | INF | INF | INF |
| **1** | INF | INF | 20 | 5 |
| **2** | 0 | INF | INF | 5 |
| **3** | 0 | INF | 5 | INF |

Redukujemy macierz ponieważ wszystkie rzędy i wiersze nie mają przynajmniej jednego zera.

| | **0** | **1** | **2** | **3** |
|---|---|---|---|---|
| **0** | INF | INF | INF | INF |
| **1** | INF | INF | 15 | 0 |
| **2** | 0 | INF | INF | 0 |
| **3** | 0 | INF | 0 | INF |

LB = 0+ 5 + 5 = 10

Następnie obliczamy koszt przejścia z 0 do 1

Cost = cost 0 + matrix[0][1] + LB = 55 + 0 +10 = 65

### 2.3. Dynamic Programming

#### 2.3.1. Dynamic Programming Teoria

Metoda ta określa ogólne podejście polegające na przekształceniu zadania optymalizacyjnego w wieloetapowy proces decyzyjny, w którym stan na każdym etapie zależy od decyzji wybieranej ze zbioru decyzji dopuszczalnych.

#### 2.3.2. Dynamic Programming implementacja algorytmu

Algorytm Helda - Karpa, zwany również algorytmem Helda - Bellmana - Karpa jest algorytmem służący do rozwiązywania problemu TSP. Jest on algorytmem dokładnym opartym na programowaniu dynamicznym. Ma on złożoność czasową $O(n^2 2n)$ oraz złożoność pamięciową równą O(nn2 ). Jest to co prawda złożoność gorsza od wielomianowej, ale algorytm ten jest znacznie lepszy od algorytmu naiwnego sprawdzającego wszystkie warianty takiego jak Brute Force. Działanie algorytmu: załóżmy, że mamy graf liczący n wierzchołków ponumerowanych 1, 2, . . . , n. Wierzchołek 1 niech będzie wierzchołkiem początkowym. Jako di,j oznaczmy odległość między wierzchołkami i oraz j (są to dane wejściowe). 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). Przykładowo, zapis D(2, 5, 6, 5) to optymalna długość ścieżki wychodzącej z punktu 1, przechodzącej przez punkty 2 i 6, kończącej się w punkcie 5. Liczbę punktów w zbiorze S oznaczmy jako s. Tym co odróżnia metrodę programowania dynamicznego od metody przeglądu zupełnego jest to, że nie musimy wyliczać odległości poszczególnych wierzchołków, a jedynie skorzystać z wcześniej policzonych wyników. W przeglądzie zupełnym każdą drogę liczymy od nowa, nie wykorzystując wcześniejszych wyliczeń, co jest tratą czasu, której unikamy w metodzie programowania dynamicznego.

Struktury wykorzystane przy implementacji:

- Hash mapa gdzie kluczem jest jest para składająca się ze ścieżki oraz wierzchołek do którego zmierzamy a informacją zapisywaną w mapie jest obecny koszt mapy oraz wierzchołek z którego przyszliśmy

## 3. Plan eksperymentu

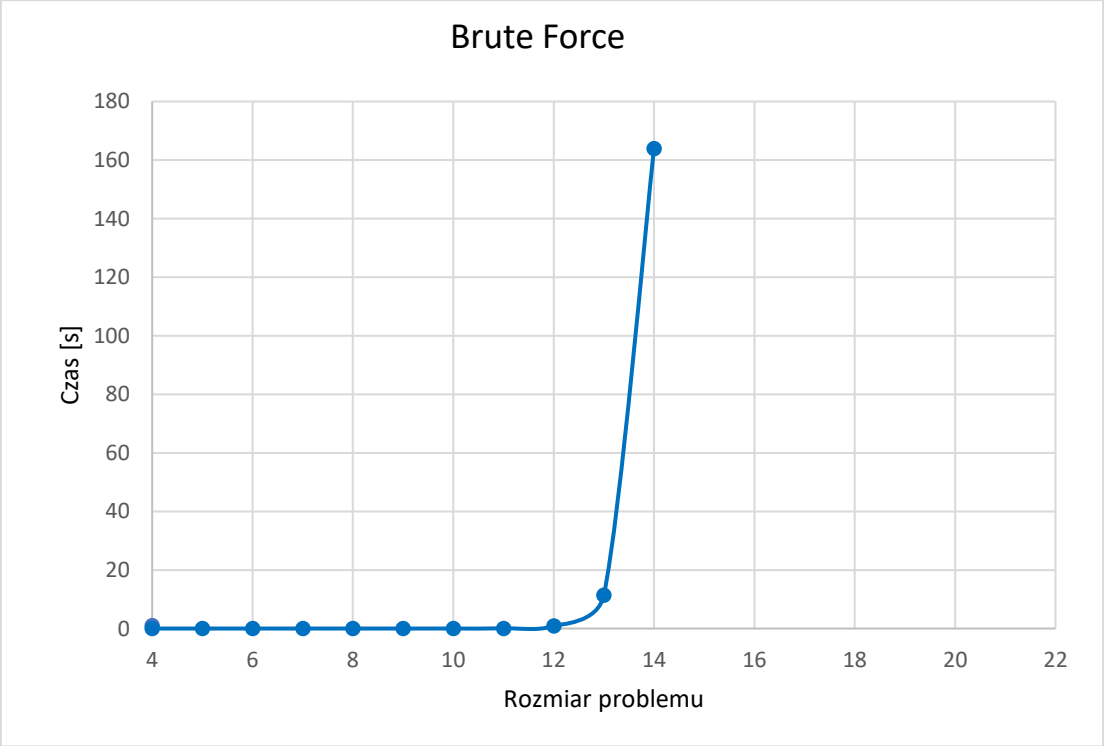
W celu wykonania eksperymentu zostały wykonane pomiary czasu wykonania każdego algorytmu. Każdy algorytm został wykonany dla różnych pod względem wielkości instancji problemu komiwojażera. Grafy wejściowe są reprezentowane w programie przez macierze sąsiedztwa, o rozmiarze od 4 do 22 wierzchołków, zawierające losowe liczby z przedziału od 1 do 100. Dane znadujące się na przekątnej sa reprezentowane jako 0. Dla każdego algorytmu o danym rozmiarze zostało wygenerowane po 100 instancji problemu w sprawozdaniu zostały podane wyniki uśrednione.

## 4. Analiza danych

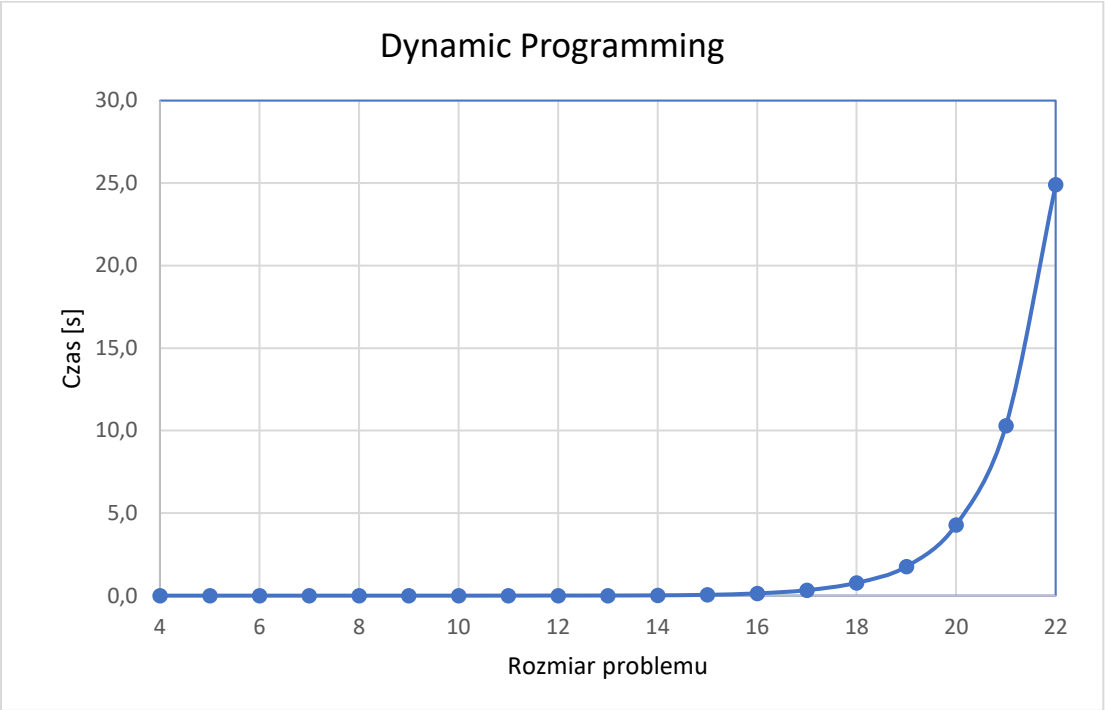
| Rozmiar Problemu | Brute Force [s] | Branch-n-Bound [s] | Branch-n-Bound [s] | Programowanie Dynamiczne [s] |
|---|---|---|---|---|
| **4** | 6,00E-07 | 2,30E-06 | 1,47E-05 | 3.3E-6 |
| **5** | 1,00E-06 | 4,60E-06 | 2,61E-05 | 9.7E-6 |
| **6** | 2,70E-06 | 1,25E-05 | 9,33E-05 | 2,62E-05 |
| **7** | 1,48E-05 | 3,80E-05 | 3,04E-04 | 8,35E-05 |
| **8** | 1,08E-04 | 1,05E-04 | 8,98E-04 | 4,68E-04 |
| **9** | 8,96E-04 | 2,78E-04 | 0,00435 | 7,35E-04 |
| **10** | 0,00706 | 5,47E-04 | 0,0123 | 0,00260 |
| **11** | 0,075 | 0,00127 | 0,021 | 0,00907 |
| **12** | 0,887 | 0,0058 | 0,0871 | 0,00649 |
| **13** | 11,4 | 0,00745 | 0,319 | 0,0274 |
| **14** | 154 | 0,0184 | 1,69 | 0,0616 |
| **15** | - | 0,0498 | - | 0,191 |
| **16** | - | 0,137 | - | 0,93 |
| **17** | - | 0,332 | - | 2,71 |
| **18** | - | 0,778 | - | 11,1 |
| **19** | - | 1,76 | - | 43,4 |
| **20** | - | 4,29 | - | - |
| **21** | - | 10,3 | - | - |
| **22** | - | 24,9 | - | - |

*Tabela 1.Uśredniony czas [s] wykonania zaimplementowanych algorytmów dla danych rozmiarów problemów.*

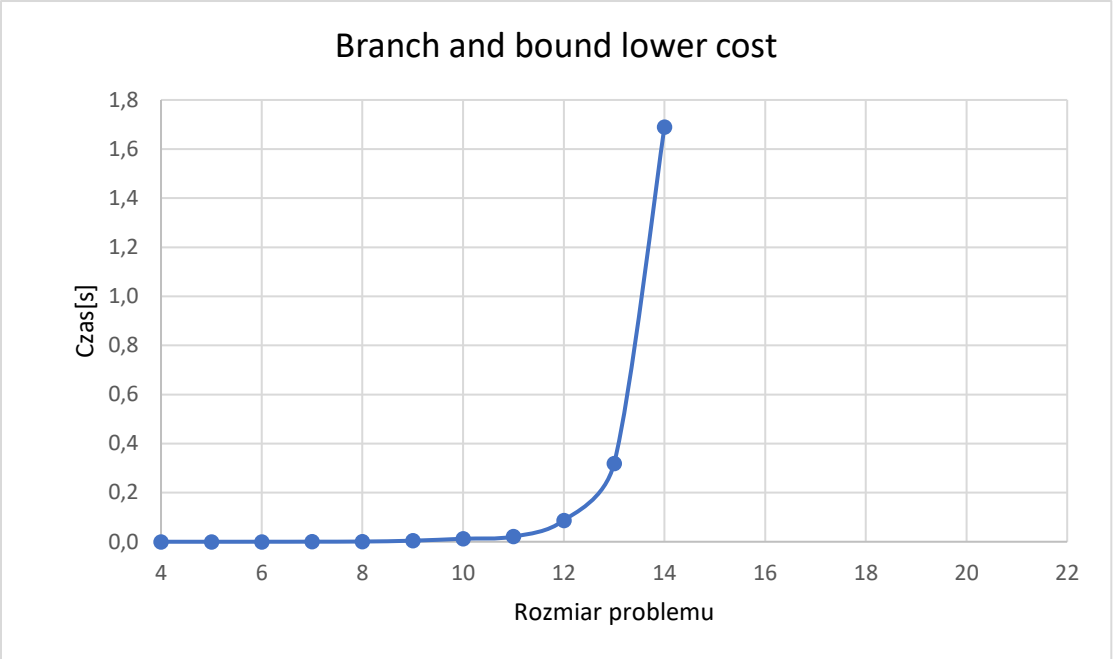
* **"-"** oznacza że czas wykonania algorytmu był zbyd długi lub komputrowi zabrakło pamięci podczas wykonywania algorytmu



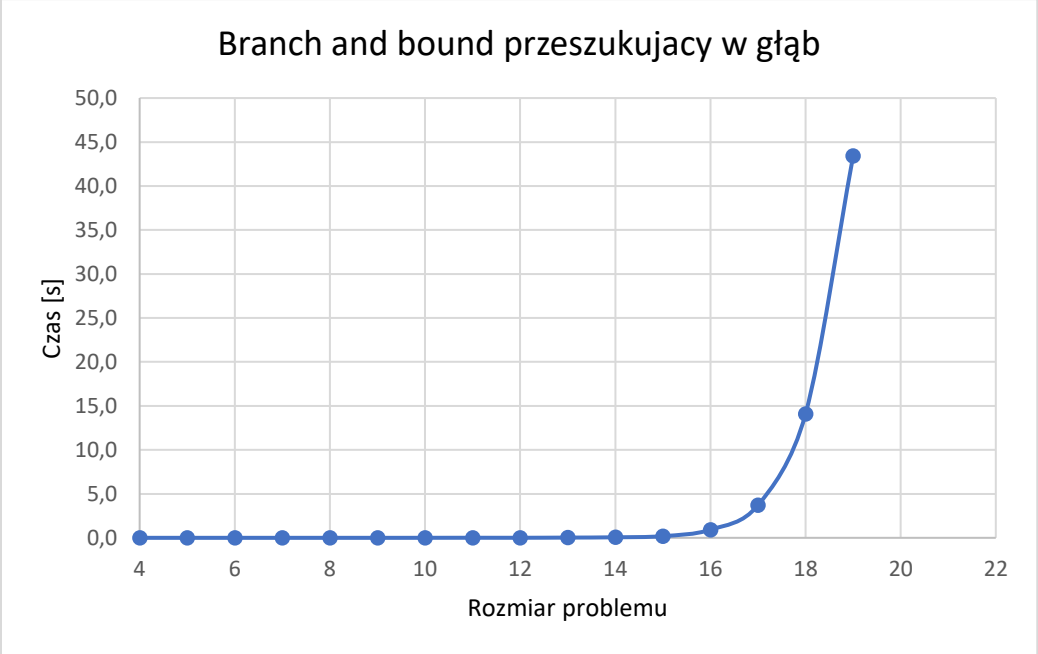
*Wykres 1.Uśredniony czas [s] wykonania algorytmu przeglądu zupełnego dla danych rozmiarów problemów.*



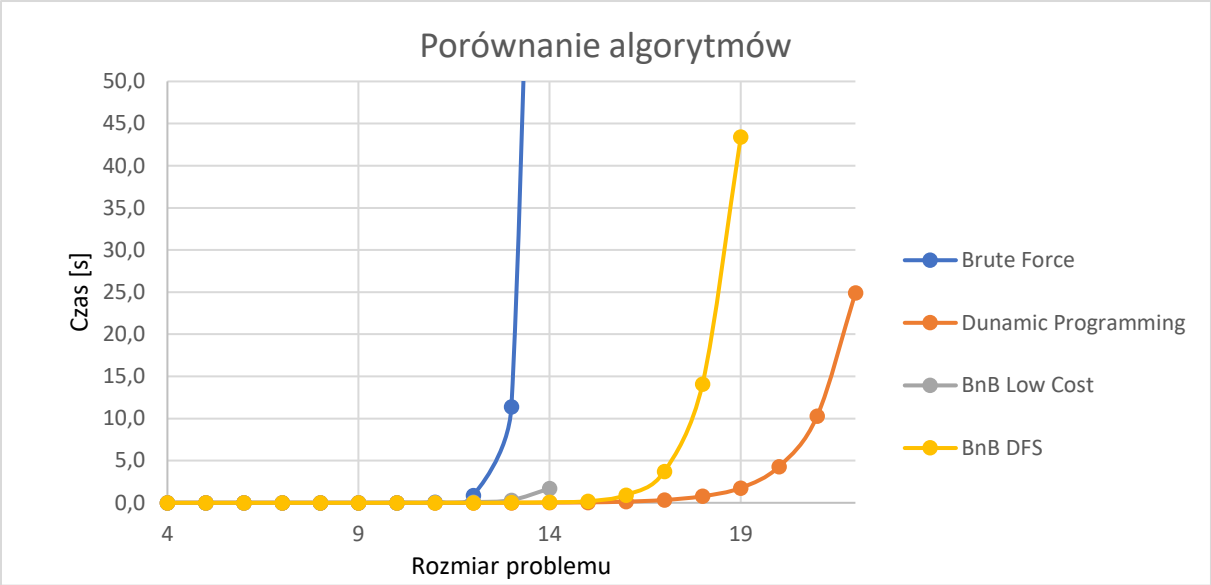

*Wykres 2.Uśredniony czas [s] wykonania algorytmu programowania dynamicznego dla danych rozmiarów problemów.*




*Wykres 3.Uśredniony czas [s] wykonania algorytmu podziału i ograniczeń dla danych rozmiarów problemów.*




*Wykres 4.Uśredniony czas [s] wykonania algorytmu podziału i ograniczeń dla danych rozmiarów problemów.*




*Wykres 5.Uśredniony czas [s] wykonania zaimplementowanych algorytmów dla danych rozmiarów problemów.*

## 5. Wnioski

Porównując uzyskane wyniki z oszacowanymi na wstępie złożonościami dla każdej metod można założyc że algortymy zostały zaprojektowne poprawnie. Algorytm przeglądu zupełnego wykazał jego złożoność obliczeniową równą O(n!). Podczas przeprowadzania eksperymentu dla metody naiwnej, przerwane zostały pomiary dla instancji problemu o wielkości do 14 wierzchołków (dla 14 wierzchołków zostało wykonane jedynie 10 prób ). Dla wiekszych wartości czas wykonania algorytmu jest nie akceptowalny. Metoda ta nadaje się jedynie do rozwiązywania instancji problemu o małych rozmiarach. Oraz dla tych przypadków jest też najszybsza. Algorytm ograniczeń i podziału w najgorszym wypadku ma też złożoność O(n!). Jednakże dzieki ograniczaniu i rozgałęzianiu czas wykonawania programu wporównaniu z brute force. Algorytmem branch and bound z przeszukiwaniem low cost jest wydajnieszy niż Brute Force jednak podczas wyknania algorytmu dla instancji problemu wiekszych niż 14 program zużywał całą pamieć, może to wynikać zbyt duzych struktór wykorystanych w algorytmie oraz w sposobie przeszukiwania. Algortym w wersji DFS działa bez rzadnych problemów. Dla problemów wiekszych niż 19 czas wykonania algorytmów stał się nie akceptowalny jednakże jest on znacznie szybszy w porównaniu z metodą naiwną. Ostatni z zaimplementowanych algorytmów wykorzystywał metodę programowania dynamicznego. Również w tym przypadku została zaobserwowana zgodność z teoretyczną złożonością obliczeniową $O(n^2 2n)$, która jest możliwa do osiągnięcia dzięki ograniczeniu działania całego algorytmu do procedury wypełniania tablicy stanów. Algorytm ten okazał się najbardziej efektywnym. Uzyskane pomiary czasów są całe rzędy niższe od tych uzyskanych w przypadku metody Branch & Bound. Zapisywanie ścieżek na jednym integerze za pomocą przesunięć bitowych oraz użycie unorderd_mapy bedącej hash mapa mającą średnią złożoność wyszukiwania O(1).

## 6. Bibliografia

1. http://www.cs.put.poznan.pl/mkasprzak/zp/ZP-wyklad5.pdf

2. https://eduinf.waw.pl/inf/alg/001_search/0140.php

3. https://pl.wikipedia.org/wiki/Problem_komiwojażera

4. https://www.techiedelight.com/travelling-salesman-problem-using-branch-and-bound/?fbclid=IwAR2XFu83vCqkwpbu93lwpfoCJO_jkRzI6YKs9GQIldRdlEam_6VWqLoJ-Ic