Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

SPRAWOZDANIE

PROJEKTOWANIE EFEKTYWNYCH ALGORYTMÓW

Implementacja i analiza efektywności algorytmu podziału i ograniczeń

Autor:
Klaudia Marzec

Prowadzący dr. inż. Dariusz Banasiak

7 listopada 2023

1 Wstęp teoretyczny

Problem komiwojażera polega na znalezieniu minimalnego cyklu Hamiltiona w pełnym grafie ważonym, czyli na poszukiwaniu w grafie takiego cyklu, który zawiera wszystkie wierzchołki (każdy dokładnie raz) i ma jak najmniejszy koszt.

Problem ten jest problemem NP- zupełnym. Założeniem projektu jest implementacja algorytmów dla asymetrycznego problemu komiwojażera- w takim grafie dla dowolnych miast A i B odległość z A do B może być inna niż odległość z B do A.

2 Przegląd zupełny (Brute Force)

2.1 Opis algorytmu

Przegląd zupełny rozwiązuje problem sprawdzając wszystkie możliwe wybory: dla problemu komiwojażera oblicza całkowitą odległość dla każdej możliwej trasy, a następnie wybiera najkrótszą. Jest to metoda nieefektywna obliczeniowo, lecz optymalna, gdyż znajduje najlepsze rozwiązanie oraz łatwa w zaimplementowaniu. W jego przypadku złożoność obliczeniowa wynosi $O(n!)$, ponieważ algorytm generuje wszystkie możliwe permutacje n miast. Tak duża złożoność obliczeniowa zdecydowanie zawyża czas działania dla dużych n.

2.2 Opis działania algorytmu

Rozpatrzmy przykład działania algorytmu krok po kroku dla przykładowej instancji o wartości N=5, przyjmując za wierzchołek startowy 2.

	0	1	2	3	4
0	0	492	738	189	442
1	515	0	41	363	513
2	766	73	0	338	663
3	176	365	298	0	673
4	477	501	699	721	0

Algorytm przegląda po kolei wszystkie możliwe rozwiązania:
$2- > 0- > 1- > 3- > 4- > 2 : 2993$
$2- > 0- > 1- > 4- > 3- > 2 : 2790$
$2- > 0- > 3- > 1- > 4- > 2 : 2532$
$2- > 0- > 3- > 4- > 1- > 2 : 2170$
$2- > 0- > 4- > 1- > 3- > 2 : 2370$
...

Po przejrzeniu wszystkich permutacji wybiera ścieżkę o najmniejszym koszcie.

2.3 Opis implementacji algorytmu

Do implementacji algorytmu została wykorzystana jedna funkcja rekurencyjna zawierająca takie parametry jak: droga, koszt, miasto, odwiedzone, start. Oznaczają one: aktualną trasę, którą ma za zadanie przejrzeć algorytm; koszt aktualnej ścieżki; aktualne miasto; licznik odwiedzonych miast; numer miasta początkowego. Początkowo sprawdzany jest warunek, czy wszystkie miasta zostały już odwiedzone.

```
void algorytm(int droga[], int koszt, int miasto, int odwiedzone, int start)
{
    if(odwiedzone == rozmiar_w)
    {
        // Powrot do miasta poczatkowego
        koszt += macierz[miasto][start];

        if(koszt < minKoszt)
        {
            minKoszt = koszt;
            for(int i=0; i<rozmiar_w; i++)
                najDroga[i] = droga[i];
        }

        for(int i=0; i<rozmiar_w; i++)
            cout << droga[i] << " -> ";

        cout << droga[rozmiar_w] << ": " << koszt << endl;

        return;
    }
```

Jeśli nie, następuje przejście do rekurencyjnego wyznaczania tras do innych miast. W pętli następuje iteracja przez kolejne miasta- jeśli miasto nie zostało jeszcze odwiedzone, to następuje aktualizacja kosztu ścieżki oraz drogi i ponowne wywołanie funkcji.

```
    for(int miasto2 = 0; miasto2 < rozmiar_w; miasto2++)
    {
        if(miasto2 == miasto)
            continue;

        if(!findd(droga, odwiedzone, miasto2))
        {
            int nowyKoszt = koszt + macierz[miasto][miasto2];
            /*if(nowyKoszt < minKoszt)
            {
                droga[odwiedzone] = miasto2;
                algorytm(droga, nowyKoszt, miasto2, odwiedzone+1);
            }*/
            droga[odwiedzone] = miasto2;
            algorytm(droga, nowyKoszt, miasto2, odwiedzone+1, start);
        }
    }
}
```

3 Metoda podziału i ograniczeń (Branch & Bound)

3.1 Opis algorytmu

Działanie metody podziału i ograniczeń opiera się na analizie drzewa przestrzeni stanów, które reprezentuje wszystkie możliwe ścieżki, jakimi może pójść algorytm rozwiązując dany problem. Przeglądanie całego drzewa byłoby bardzo kosztowne ze względu na jego wykładniczy rozmiar, dlatego metoda ta w każdym węźle oblicza granicę (ograniczenie), która pozwala określić go jako obiecujący bądź nie. W dalszej fazie algorytm przegląda tylko potomków węzłów obiecujących. Pozwala to zmniejszyć ilość odwiedzanych wierzchołków i szybciej znaleźć rozwiązanie problemu.

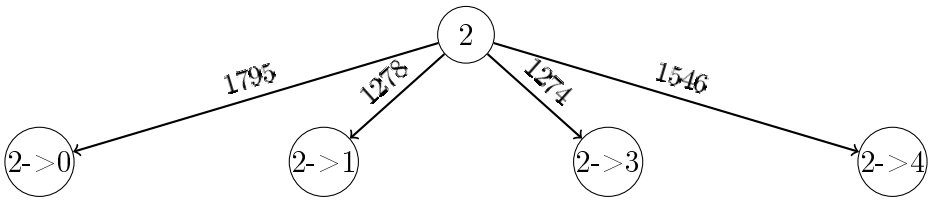
3.2 Opis działania algorytmu

Rozpatrzmy przykład działania algorytmu krok po kroku dla przykładowej instancji o wartości N=5, przyjmując za wierzchołek startowy 2.

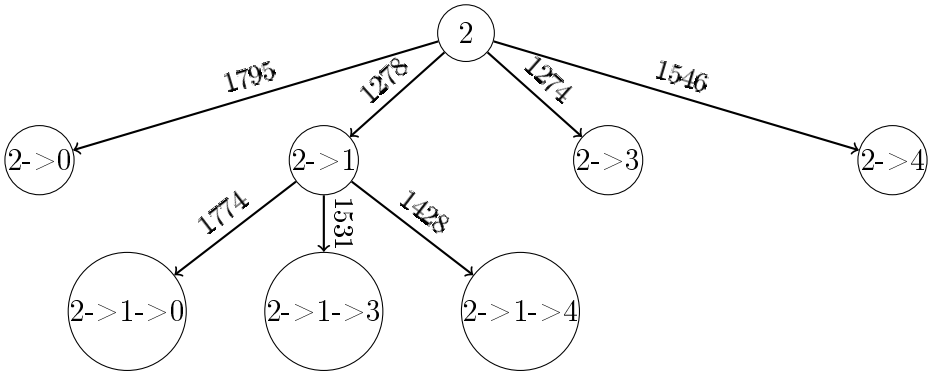
	0	1	2	3	4
0	0	492	738	189	442
1	515	0	41	363	513
2	766	73	0	338	663
3	176	365	298	0	673
4	477	501	699	721	0

Drzewo przestrzeni stanów zdefiniowane jest jako drzewo, które w korzeniu zawiera liste zawierającą tylko wierzchołek początkowy. Na pierwszym poziomie znajdują się węzły zawierające listy dwóch wierchołków, na drugim trzech itd. Liście będą zawierać listy (n-1)- elementowe, gdyż wtedy ostatni wierzchołek trasy jest już wyznaczany jednoznacznie. Początkowo w drzewie znajduje się tylko korzeń

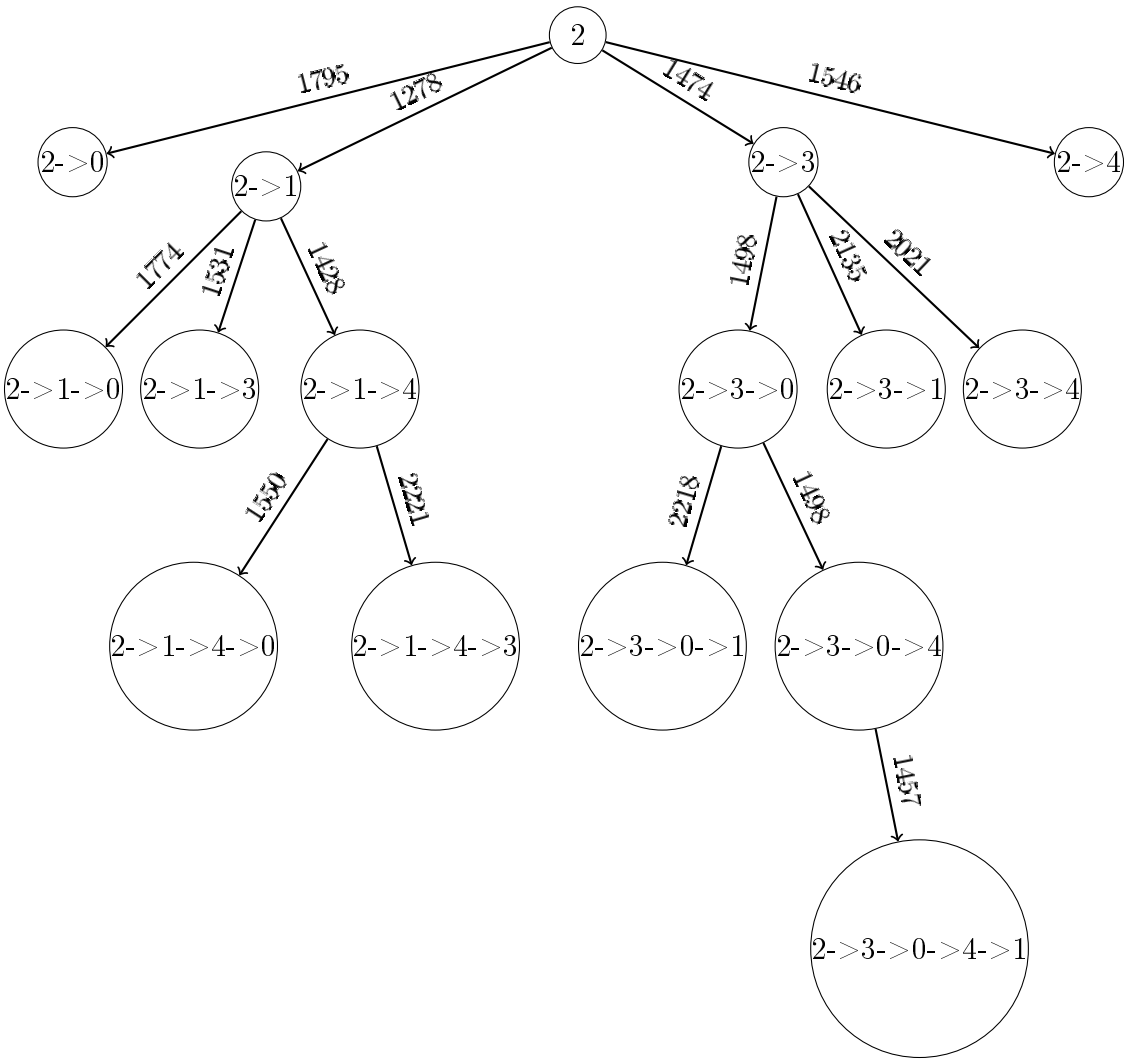
W kolejnej fazie algorytm przegląda potomków korzenia, oblicza ich granicę oraz zaznacza wszystkie wierzchołki jako obiecujące, ponieważ aktualnie najmniejszy koszt nie został jeszcze znaleziony (domyślnie jego wartość na początku to ∞). Wierzchołki obiecujące dodane zostają do kolejki priorytetowej.



Następnie algorytm wybiera wierzchołek o najlepszej granicy i rozszerza jego potmków.



W kolejnych fazach algorytm powtarza swoje działanie- ponownie szuka wierzchołka o najlepszej granicy i rozwija jego potomków, aż do uzyskania całej ścieżki. Na kolejnym rysunku przedstawiono finalny wygląd drzewa po wykonaniu całego algorytmu.



3.3 Opis implementacji algorytmu

Wybór strategii przeszukiwania grafu

W celu zwiększenia efektywności działania algorytmu, zastosowana została strategia odwiedzania wierzchołków "Najpierw najlepszy". Po przejrzeniu synów jakiegoś węzła przeglądane są wszystkie pozostałe nierozwinięte obiecujące węzły i rozwijany zostaje węzeł o najlepszej granicy. Do prawidłowego zaimplementowania wybranej strategii została wykorzystana kolejka priorytetowa, która została stworzona jako nowa klasa zawierająca wskaźnik na początek tablicy elementów typu Wierzchołek. Struktura Wierzchołek została stworzona w celu zdefiniowania pojedynczego węzła w drzewie, zawierającego takie wartości jak: trasa, poziom drzewa, granica oraz koszt. Klasa KolejkaPriorytetowa zawiera metody ułatwiające działanie algorytmu takie jak dodawanie nowego wierzchołka do kolejki, usuwanie danego wierzchołka oraz znalezienie najlepszego wierzchołka.

```
3
4   struct Wierzcholek
5   {
6       int *trasa;
7       int poziom;
8       int granica;
9       int koszt;
10  };
```

Funkcja obliczająca granicę

Żeby obliczyć granicę węzła należało określić dolną granicę długości ścieżek, których prefiksem jest ścieżka w obecnym węźle.

- Długość trasy nie może być krótsza niż suma minimalnych długości krawędzi wychodzących z każdego wierzchołka
- Minimalną długość krawędzi wychodzących z wierzchołka i możemy wyznaczyć przeglądając i-ty wiersz macierzy sąsiedztwa

Korzystając z powyższych spostrzeżeń funkcja obliczjąca granicę została wyznaczona w przedstawiony sposób: dla każdego wierzchołka jest obliczana minimalna długość krawędzi wychodzących z niego, które jeszcze mogą być użyte. Dla wierzchołków znajdujących się w częściowym rozwiązaniu (poza ostatnim na tej częściowej trasie) znane są wartośći krawędzi wychodzących. Dla pozostałych wierzchołków:

- Brane są pod uwagę wszystkie krawędzie nie prowadzące do wierzchołków umieszczonych w częsciowym rozwiązaniu
- Jeśli to nie jest ostatni wierzchołek w częściowym rozwiązaniu, brane są pod uwagę krawędzie prowadzące do wierzchołka początkowego

Mając obliczone minimalne długości krawędzi wychodzących z każdego wierzchołka obliczamy ich sumę, która będzie wartością funkcji obliczjącej granicę. Nie ma pewności, że istnieje trasa o takiej długości, lecz na pewno nie ma trasy o długości mniejszj.

```
int minDlugosc(int wierzcholek, int *droga, int roz)
{
    int minimum = INT_MAX;

    for(int i=0; i<rozmiar; i++)
    {
        int blad = 0;

        if(wierzcholek == droga[roz-1])
        {
            for(int j=0; j<roz; j++)
            {
                if(droga[j] == i)
                    blad = 1;
            }
        }
        else
        {
            for(int j=1; j<roz; j++)
            {
                if(droga[j] == i)
                    blad = 1;
            }
        }

        if(blad == 1 || i == wierzcholek)
            continue;

        if(macierz[wierzcholek][i] < minimum)
            minimum = macierz[wierzcholek][i];
    }

    return minimum;
}
```

```
int bound(int *wierzcholek, int poziom)
{
    int roz = poziom+1;

    // Przekazujemy tablice miast - trasa w danym wierzcholku grafu

    int sumaMinimum = 0;

    // Dlugosc tras pomiedzy miastami w wierzcholku

    for(int i=0; i<roz-1; i++)
        sumaMinimum += macierz[wierzcholek[i]][wierzcholek[i+1]];

    if(roz==rozmiar)
    {
        return sumaMinimum;
    }

    // Minimum sposrod pozostalych

    for(int i=0; i<rozmiar; i++)
    {
        int blad = 0;
        for(int j=0; j<roz-1; j++)        // Sprawdzanie wierzcholkow
        {
            if(wierzcholek[j] == i)
                blad = 1;
        }

        if(blad == 1)
            continue;

        // Wyznaczanie minimum dla kolejnych wierzcholkow

        int minDl = minDlugosc(i, wierzcholek, roz);
        sumaMinimum += minDl;
    }

    return sumaMinimum;
}

void branchBound()
{
```

4 Plan eksperymentu

W celu sprawdzenia poprawności działania algorytmów, dokonano pomiaru czasu działania w zależności od rozmiaru problemu N. Testy efektywności poszczególnych algorytmów zostały przeprowadzone na losowo wygenerowanych grafach o rozmiarach N: 5, 6, 7, 8, 9, 10, 11, 12, 13. Każdy pomiar został powtórzony 100 razy.

Generowanie grafu

Każdy graf, na którym były wykonane pomiary, został losowo wygenerowany za pomocą odpowiedniej funkcji uzupełniającej strukturę grafu losowymi wartościami, lecz uzuepłniającą go w taki sposób, aby odległości między miastami A->B oraz B->A miały wartości zbliżone do siebie z maksymalną różnicą 50.

```
        {
            // Wylosowanie odleglosci zblisonej do tej ktora je

            int ind = graf.znalezienie(j, i);
            int odleglosc = graf.krawedzie[ind].przepustowosc;

            if(i==j)
                k.przepustowosc = 0;
            else
            {
                // Losowanie znaku + lub -

                int znak = rand() % 2;

                // 0 -
                // 1 +

                // Losowanie roznicy

                int roznica = rand() % 51;

                if(znak == 0)
                    k.przepustowosc = odleglosc - roznica;
                else
                    k.przepustowosc = odleglosc + roznica;
            }
        }
        else
        {
            // Losowanie odleglosci miedzy miastami

            if(i==j)
                k.przepustowosc = 0;
            else
            {
                k.przepustowosc = rand() % 1000 + 1;          //
            }
        }

        graf.completeKrawedzie(index++, k);
    }
}

return graf;
```

Pomiar czasu

Do pomiarów czasu została wykorzystana operacja QueryPerformanceCounter- start przed wywołaniem danej funkcji, koniec zaaraz po zakończeniu jej działania. Wyniki były zapisywane w pliku tekstowym w milisekundach.

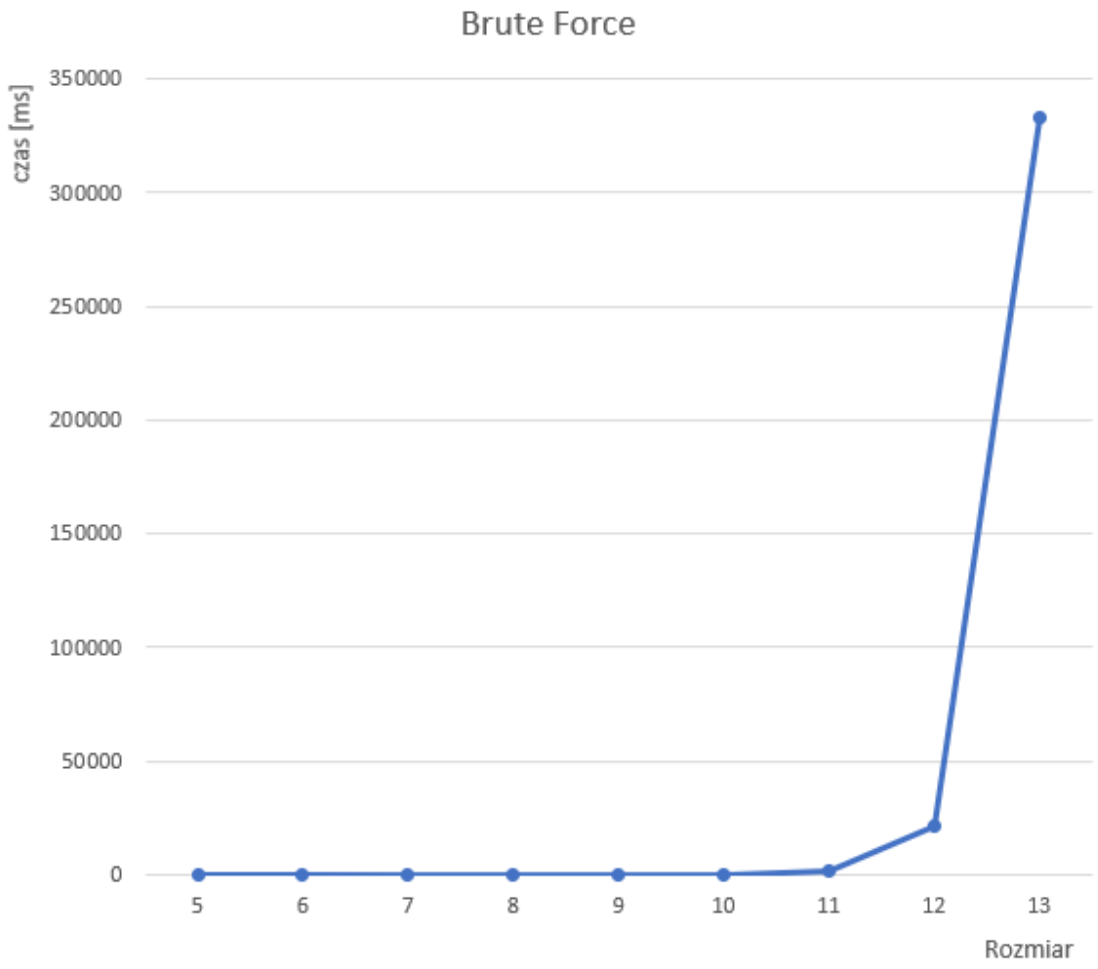
5 Wyniki

Wyniki w tabeli zostały przedstawione w milisekundach. Tabela przedstawia uśrednione wyniki dla każdego rozmiaru dla dwóch algorytmów- Brute Force oraz Branch And Bound.

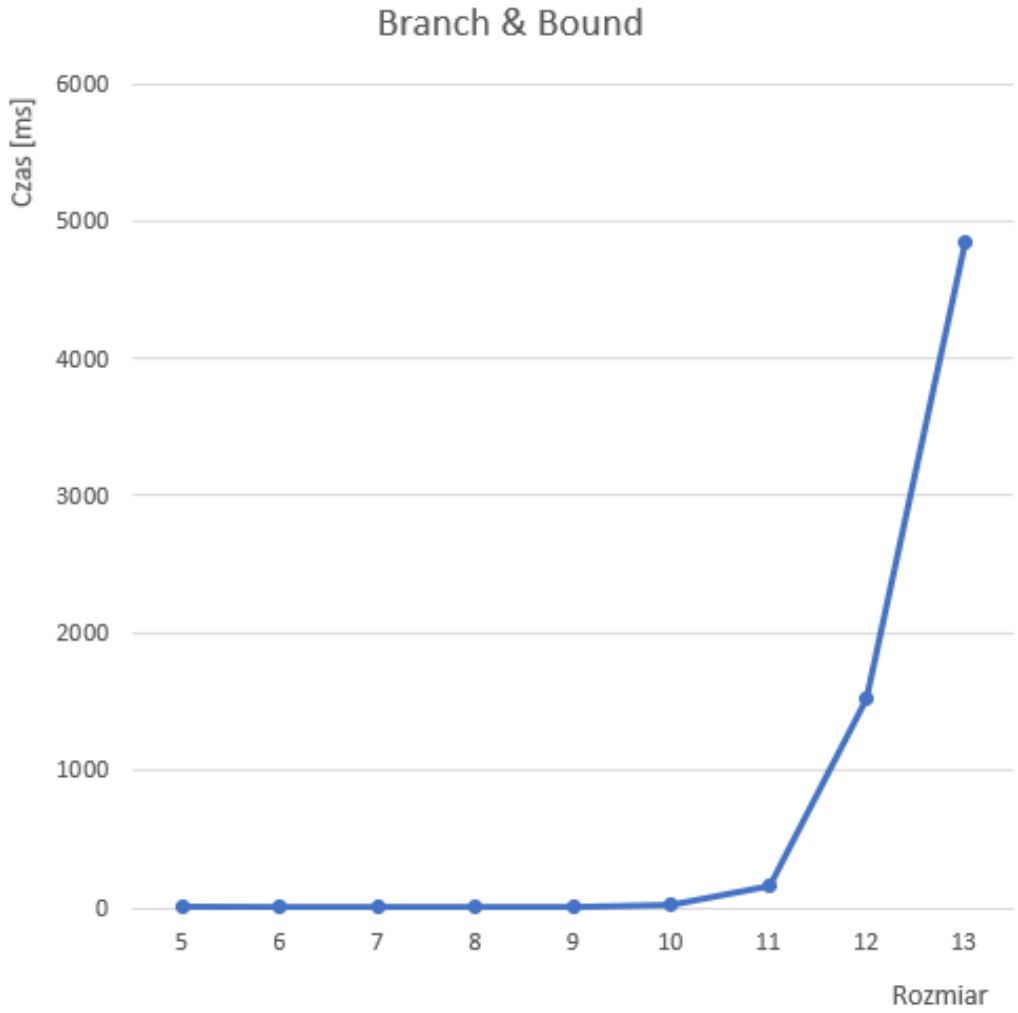
N	Brute Force [ms]	B&B [ms]
5	0.00537	0.305753
6	0.027078	0.426569
7	0.190713	1.372449
8	1.478843	4.239863
9	15.000451	5.806272
10	142.22986	26.516608
11	1546.338	160.181694
12	21811.187	1528.482828
13	332665.8696	4843.322433

WYKRESY

Wykres algorytmu Brute Force

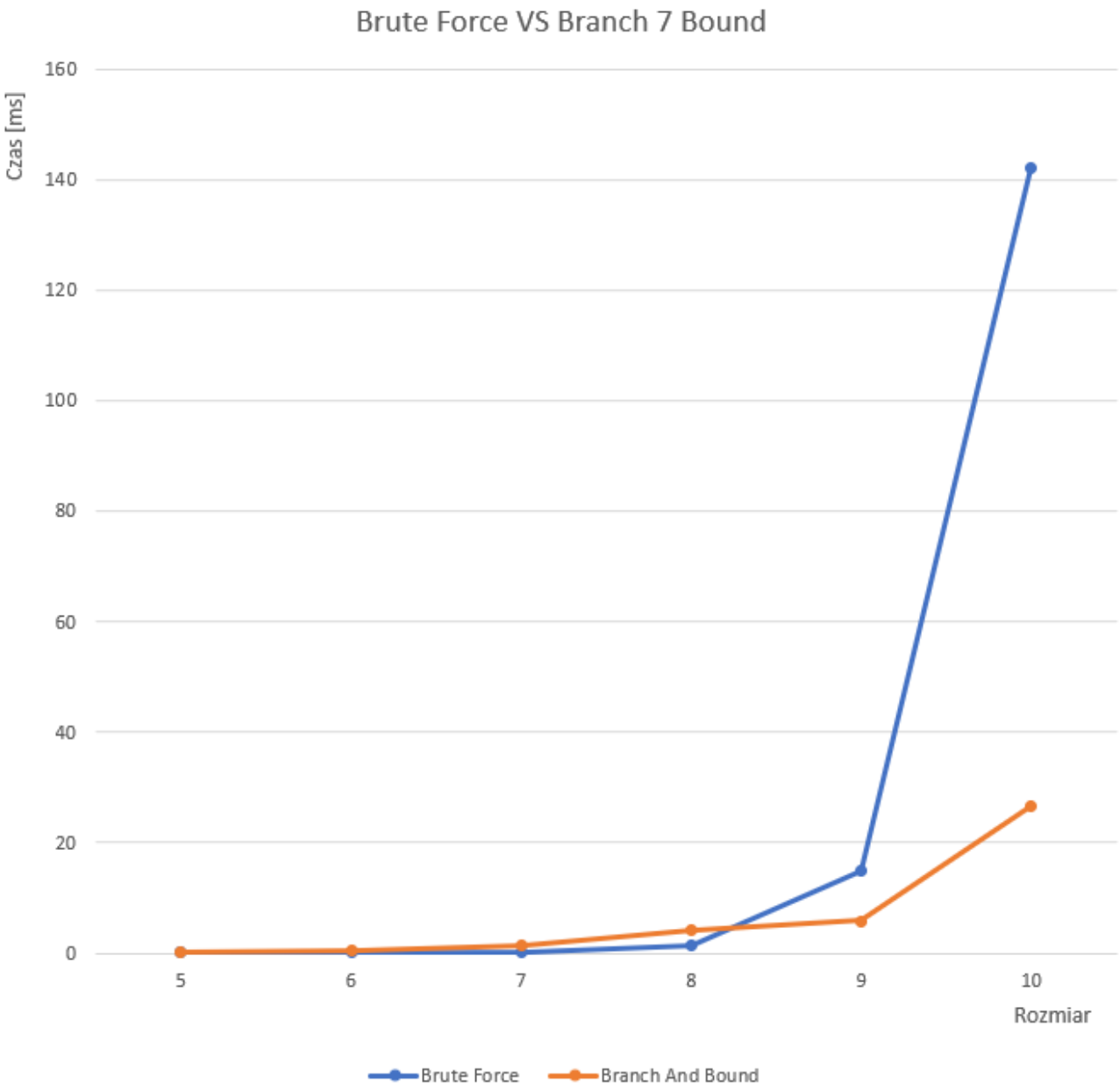


Wykres algorytmu Branch And Bound



W celu lepszego ukazania porównania między algorytmami wybrano zakres rozmiaru grafu w przedziale 5-10

Wykres porównujący oba algrytmy dla rozmiarów 5-10



6 Wnioski

Z wykresów można zauważyć, że dla małych instancji problemu algorytm Brute Force jest minimalnie szybszy od algorytmu Branch & Bound. Przykładowo dla grafu o rozmiarze 6 średni czas działania algorytmu to 0.02 ms, natomiast dla B&B jest to 0.43 ms, czyli zdecydowanie dłużej, jednak różnica nie jest zauważalna dla człowieka, ponieważ są to wyniki rzędu 10^{-4} sekundy.

Sytuacja ulega zmianie dla grafów o większym rozmiarze. Na wykresie jest to bardzo dobrze widoczne od instancji o rozmiarze 9. W tym przypadku średni czas algorytmu Brute Force to 15 ms, zaś Branch & Bound 5.8 ms, czyli już trzykrotnie krócej. Dla coraz większych rozmiarów różnica jest coraz większa, co można zaobserwować na podstawie wyników z tabeli. Największa zbadana instancja wynosi 13. Dla niej algorytm Brute Force trwa juz śrenio 333.665 s czyli około 5.5 minuty. Algorytm Branch & Bound trwa około 4.843 sekundy- różnica jest zatem diametralna.

Otrzymane wyniki są zgodne z założeniami- metoda podziału i ograniczeń jest zdecydowanie bardziej efektywna i praktyczna niż metoda przeszukiwania zupełnego dla problemu komiwojażera. Choć algorytm Brute Force jest dokładny, to jego złożoność obliczeniowa stawia go poza zasięgiem dla większych instancji problemu.

7 Kod źródłowy

https://github.com/Klaudiaamarzec/Branch-Bound

8 Bibliografia

- https://pl.wikipedia.org/wiki/Problem_komiwoja%C5%BCera
- http://algorytmy.ency.pl/artykul/problem_komiwojazera
- https://ii.uni.wroc.pl// prz/2011lato/ah/opracowania/met_podz_ogr.opr.pdf