Politechnika Wrocławska

Projektowanie efektywnych algorytmów – projekt nr 1

Autor:

Łukasz Gawron,
nr indeksu 264475

Termin oddania projektu:

20.11.2023

Termin zajęć:

Poniedziałek godz. 15:15

Prowadzący:

Dr. Inż. Jarosław Mierzwa

Spis treści

1. Wstęp

Celem projektu była implementacja oraz zbadanie efektywności algorytmów dokładnych dla problemu Komiwojażera – zagadnieniu z teorii grafów, które polega na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Badany wariant problemu to ATSP (asymetryczny problem komiwojażera) – co oznacza, że waga ścieżki z wierzchołka A do wierzchołka B jest różna od tej z wierzchołka B do wierzchołka A. Główną trudnością problemu jest duża liczba danych do analizy. W przypadku asymetrycznego problemu komiwojażera dla n miast liczba kombinacji wynosi $(n-1)!$. Problem Komiwojażera jest problemem klasy NP (o wielomianowym czasie rozwiązywania).

1.1 Badane algorytmy oraz ich złożoność

1.1.1 Brute Force (Przegląd zupełny)

W przypadku algorytmu Brute Force należy zbadać wszystkie możliwe permutacje wierzchołków – dokonując przeglądu zupełnego grafu, złożoność czasowa dla tego algorytmu wynosi $O(n!)$, co oznacza, że algorytm ten staje się niepraktyczny już dla grafów o kilkunastu wierzchołkach. Największą strukturą, używaną w przypadku tego algorytmu stanowi sama macierz sąsiedztwa grafu, co daje złożoność pamięciową na poziomie $O(n^2)$.

1.1.2 Branch and Bound (algorytm Little'a)

Zasada działania algorytmu Little'a polega na wybieraniu krawędzi pomiędzy wierzchołkami, które powodują najmniejszy wzrost dolnego ograniczenia dla rozwiązań niezawierających tej krawędzi, po czym następuje podział na dwa poddrzewa – zawierające i niezawierające wybranej krawędzi. Oba poddrzewa zostają następnie dodane do kolejki priorytetowej (sortowanej malejąco według dolnego ograniczenia). Dokonujemy owego podziału problemu dla danego grafu, aż nie osiągniemy macierzy o wymiarze 2 na 2 – wtedy wybieramy 2 ostatnie krawędzie w taki sposób, aby tworzyły one kompletną ścieżkę. Wynika z tego, że optymistyczna złożoność czasowa zbliżona jest do funkcji wykładniczej (nieco lepsza od $O(2^n)$). Pesymistyczny przypadek natomiast, dla grafu, w którym poddrzewa będą posiadały to samo dolne ograniczenie, wymusza powrót do poprzednich węzłów drzewa i ich dalszy podział. W najgorszym wypadku należałoby sprawdzić wszystkie możliwe poddrzewa, co daje złożoność czasową identyczną, jak w przypadku algorytmu Brute Force, czyli $O(n!)$.

Największą strukturą używaną w implementacji algorytmu jest kolejka priorytetowa, przechowująca niezbadane gałęzie drzewa, reprezentowane za pomocą macierzy, które same w sobie posiadają złożoność pamięciową $O(n^2)$. Całkowita złożoność pamięciowa zależy od szybkości znalezienia rozwiązania (niekoniecznie najlepszego – rozwiązanie pozwala na odrzucenie tych gałęzi, których dolne ograniczenie jest wyższe). Jednakże, z powodu, iż przy algorytmie Little'a podział dokonujemy według wyboru krawędzi, których w grafie pełnym jest $(n \cdot (n-1))/2$, a strategia wyboru kolejnych węzłów do podziału polega na wyborze węzłów o minimalnym ograniczeniu, może to doprowadzić do nagromadzenia się dużej ilości węzłów w kolejce o złożoności pamięciowej $O(n^2)$ każdy.

2. Opis działania algorytmu na przykładzie

2.1 Algorytm Little'a

1. Redukcja macierzy wejściowej

Niech dany będzie graf opisany macierzą wejściową. Indeksy wierszy oraz kolumn będą liczone od 1:

$$\begin{bmatrix} \infty & 20 & 30 & 31 & 28 & 40 \\ 30 & \infty & 10 & 14 & 20 & 44 \\ 40 & 20 & \infty & 10 & 22 & 50 \\ 41 & 24 & 20 & \infty & 14 & 42 \\ 38 & 30 & 32 & 24 & \infty & 28 \\ 50 & 54 & 60 & 52 & 38 & \infty \end{bmatrix}$$

W pierwszym kroku należy zredukować macierz wejściową, zaczynając od wierszy (od góry), następnie kolumny (od lewej).

$$Koszt\ redukcji\ względem\ wierzy\ =\ 20\ +\ 10 + 10 + 14 + 24 + 38\ =\ 116$$

$$\begin{bmatrix} \infty & 0 & 10 & 11 & 8 & 20 \\ 20 & \infty & 0 & 4 & 10 & 34 \\ 30 & 10 & \infty & 0 & 12 & 40 \\ 27 & 10 & 6 & \infty & 0 & 28 \\ 14 & 6 & 8 & 0 & \infty & 4 \\ 12 & 16 & 22 & 14 & 0 & \infty \end{bmatrix}$$

$$Koszt\ redukcji\ względem\ kolumn\ (\text{kulumna nr 1 oraz nr 6})\ =\ 12\ +\ 4 = 16$$

$$\begin{bmatrix} \infty & 0 & 10 & 11 & 8 & 16 \\ 8 & \infty & 0 & 4 & 10 & 30 \\ 18 & 10 & \infty & 0 & 12 & 36 \\ 15 & 10 & 6 & \infty & 0 & 24 \\ 2 & 6 & 8 & 0 & \infty & 0 \\ 0 & 16 & 22 & 14 & 0 & \infty \end{bmatrix}$$

$$Dolne\ ograniczenie\ =\ 116\ +\ 16\ =\ 132$$

2. Wybór łuku

Następnym krokiem jest wybór łuku, który spowoduje największy wzrost dolnego ograniczenia dla rozwiązań niezawierających tego łuku. Kandydaci na wybór łuku to kolejno:

$$(1,2): 8 + 6 = 14$$

$$(2,3): 4 + 6 = 10$$

$$(3,4): 10 + 0 = 10$$

$$(4,5): 6 + 0 = 6$$

$$(5,4) = 0 + 0 = 0$$

$$(5,6) = 0 + 16 = 16$$

$$(6,1) = 0 + 2 = 2$$

$$(6,5) = 0 + 0 = 0$$

Wybieramy łuk $(5,6)$ i dokonujemy podziału na dwa poddrzewa, jedno zawierające krawędź, drugie nie zawierające tej krawędzi. Wybór łuku w dalszej części przykładu będzie dokonywany w sposób analogiczny.

3. Podział na dwa pod-drzewa

Lewa macierz poddrzewa będzie wyglądała w sposób następujący:

$$\begin{bmatrix} \infty & 0 & 10 & 11 & 8 & \infty \\ 8 & \infty & 0 & 4 & 10 & \infty \\ 18 & 10 & \infty & 0 & 12 & \infty \\ 15 & 10 & 6 & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ 0 & 16 & 22 & 14 & \infty & \infty \end{bmatrix}$$

$$Dolne\ ograniczenie\ = 132$$

Wykreślone zostały wiersz nr 5 oraz kolumna nr 6, oraz „powrót” z wierzchołka 6 do wierzchołka 5 (macierz [6][5]). Jak widzimy, w każdym wierszu oraz w każdej kolumnie, które nie zostały jeszcze wykreślone – pozostały jakieś 0, zatem koszt redukcji tej macierzy wynosi 0 (nie redukujemy macierzy). Dolne ograniczenie pozostaje na poziomie 132.

W prawej macierzy wykluczamy wybór łuku $(5,6)$ przez wstawienie tam nieskończoności, po czym należy sprawdzić, czy macierzy nie można zredukować. Zredukować należy jedynie kolumnę nr 6.

$$\begin{bmatrix} \infty & 0 & 10 & 11 & 8 & 0 \\ 8 & \infty & 0 & 4 & 10 & 14 \\ 18 & 10 & \infty & 0 & 12 & 20 \\ 15 & 10 & 6 & \infty & 0 & 8 \\ 2 & 6 & 8 & 0 & \infty & \infty \\ 0 & 16 & 22 & 14 & 0 & \infty \end{bmatrix}$$

$$Dolne\ ograniczenie\ =\ 132 + 16 = 148$$

Po dokonaniu podziału na poddrzewa, wybieramy gałąź z najmniejszym dolnym ograniczeniem, po czym ponownie wybieramy łuk, po którym będziemy dokonywać podziału. Powtarzamy tę czynność, dopóki macierz nie będzie w postaci 2x2 (pozostałe wiersze i kolumny macierzy wykreślone poprzez wpisanie na całej długości wartości nieskończoności).

4. Podział macierzy z LB = 132, wybrane łuki: (5,6)

$$\begin{bmatrix} \infty & 0 & 10 & 11 & 8 & \infty \\ 8 & \infty & 0 & 4 & 10 & \infty \\ 18 & 10 & \infty & 0 & 12 & \infty \\ 15 & 10 & 6 & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ 0 & 16 & 22 & 14 & \infty & \infty \end{bmatrix}$$

Wybieramy łuk $(6,1)$: $14 + 8\ =\ 22$

Dokonujemy podziału na macierz lewą, zawierającą łuk (6,1), wykreślany powrót z 1 do 6 (już znajdywała się tam nieskończoność przez wcześniejsze operacje). Dodatkowo widzimy, że wybrane krawędzie tworzą pod-ścieżkę $5 \rightarrow 6 \rightarrow 1$, więc należy wykluczyć powrót z 1 do 5 poprzez wpisanie tam nieskończoności. Koszt redukcji macierzy po wyborze łuku wynosi 0:

$$\begin{bmatrix} \infty & 0 & 10 & 11 & \infty & \infty \\ \infty & \infty & 0 & 4 & 10 & \infty \\ \infty & 10 & \infty & 0 & 12 & \infty \\ \infty & 10 & 6 & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

$$Dolne\ ograniczenie\ =\ 132\ +\ 0 = 132$$

Prawa macierz tego pod-drzewa będzie wyglądała natomiast następująco (wykreślamy łuk (6,1) po czym redukujemy wiersz nr 6 oraz kolumnę nr 1).

Koszt redukcji macierzy to $14\ (wiersze) + 8\ (\text{kolumny})$:

$$\begin{bmatrix} \infty & 0 & 10 & 11 & 8 & \infty \\ 0 & \infty & 0 & 4 & 10 & \infty \\ 10 & 10 & \infty & 0 & 12 & \infty \\ 7 & 10 & 6 & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & 2 & 8 & 0 & \infty & \infty \end{bmatrix}$$

$$Dolne\ ograniczenie\ =\ 132\ +\ 22\ =\ 154$$

5. Podział macierzy z LB = 132, wybrane łuki: (5,6), (6,1)

$$\begin{bmatrix} \infty & 0 & 10 & 11 & \infty & \infty \\ \infty & \infty & 0 & 4 & 10 & \infty \\ \infty & 10 & \infty & 0 & 12 & \infty \\ \infty & 10 & 6 & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

Wybieramy łuk $(1,2)$: $10 + 10 = 20$

Dokonujemy podziału na macierz lewą, zawierającą łuk (1,2) oraz blokujemy powrót z wierzchołka 2 do 1, dodatkowo należałoby zablokować powstawanie pomniejszych cykli z niepełnej ścieżki, jaką tworzą wybrane łuki: $5 \rightarrow 6 \rightarrow 1 \rightarrow 2$, jednak w tym wypadku ścieżki te są już „wykreślone”, więc nie musimy wykonywać żadnej dodatkowej akcji. Koszt redukcji lewej macierzy wynosi 0.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 4 & \infty & \infty \\ \infty & \infty & \infty & 0 & 12 & \infty \\ \infty & \infty & 6 & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

$$Dolne\ ograniczenie = 132 + 0 = 132$$

Prawa macierz pod-drzewa będzie natomiast wyglądała w sposób następujący (wykreślamy łuk (1,2), po czym należy zredukować wiersz nr 1 oraz kolumnę nr 2). Koszty redukcji prawej macierzy to $10\ (wiersze) +\ 10\ (kolumny)$:

$$\begin{bmatrix} \infty & \infty & 0 & 1 & \infty & \infty \\ \infty & \infty & 0 & 4 & 10 & \infty \\ \infty & 0 & \infty & 0 & 12 & \infty \\ \infty & 0 & 6 & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

$$Dolne\ ograniczenie\ =\ 132\ +\ 20 = 152$$

6. Podział macierzy z LB = 132, wybrane łuki: (5,6), (6,1), (1,2)

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 4 & \infty & \infty \\ \infty & \infty & \infty & 0 & 12 & \infty \\ \infty & \infty & 6 & \infty & 0 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

Wybieramy łuk $(4,5)$: $6 + 12 = 18$

Dokonujemy podziału na macierz lewą, zawierającą łuk (4,5), wykreślając odpowiednio wiersz oraz kolumnę. Wybrane krawędzie tworzą pod-ścieżkę: $4 \rightarrow 5 \rightarrow 6 \rightarrow 1 \rightarrow 2$, zatem należy zablokować przejście z wierzchołka 2 do 4. Lewa macierz wygląda zatem następująco. Koszt redukcji lewej macierzy wynosi 0:

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

$$Dolne\ ograniczenie\ =\ 132\ +\ 0 = 132$$

Prawa macierz pod-drzewa będzie natomiast wyglądała w sposób następujący (wykreślamy łuk (4,5) po czym należy zredukować wiersz nr 4 oraz kolumnę nr 5).

Koszty redukcji to $6\ (wiersze)\ +\ 12\ (kolumny)$:

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & 4 & \infty & \infty \\ \infty & \infty & \infty & 0 & 0 & \infty \\ \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

$$Dolne\ ograniczenie\ =\ 132\ +\ 18 = 150$$

7. Wybór łuków macierzy 2x2 z LB 132, wybrane łuki: (5,6), (6,1), (1,2), (4,5)

Ostatnim etapem rozgałęziania drzewa jest wybór dwóch ostatnich łuków, w ten sposób, aby utworzyć kompletny cykl Hamiltona. W tym wypadku pozostały nam do wyboru jedynie 2 łuki, jednak zdarza się, że w macierzy okazji mamy 3 łuki do wyboru.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty & \infty \\ \infty & \infty & \infty & 0 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

Wybieramy łuki:

$$(2,3): koszt\ =\ 0$$

$$(3,4): koszt\ =\ 0$$

$$Dolne\ ograniczenie\ =\ 132 + 0 + 0$$

Otrzymujemy w ten sposób pierwsze rozwiązanie:

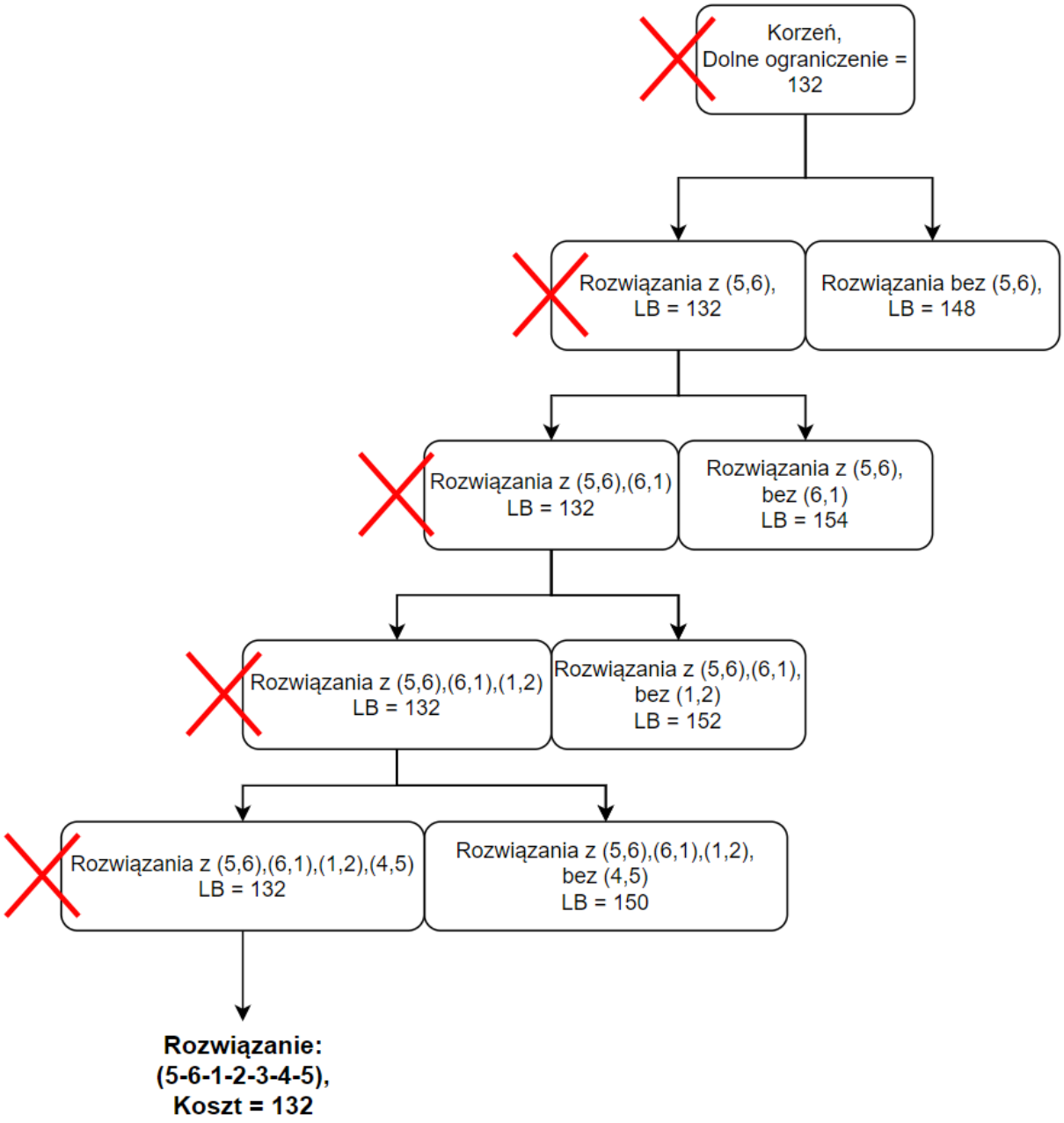
- Koszt ścieżki: 132
- Krawędzie: (5,6), (6,1), (1,2), (4,5), (2,3), (3,4)
- Ścieżka: $5 \rightarrow 6 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$

8. Sprawdzenie pozostałych węzłów drzewa

Kontynuujemy sprawdzanie dostępnych węzłów drzewa pozostałych w kolejce. Należy sprawdzić, czy występuje odgałęzienie o niższym dolnym ograniczeniu.

Nasze drzewo prezentuje się w sposób następujący:

- Czerwonym znakiem X oznaczone zostały węzły, wykorzystane do dokonania podziału, więc logicznym jest, że zostały usunięte z kolejki. Do wyboru mamy jedynie węzły bez tego oznaczenia.



Jak widzimy, żadne z węzłów, które pozostają w kolejce, nie posiadają ograniczenia LB mniejszego od dotychczasowego rozwiązania, więc wszystkie pozostałe rozgałęzienia zostają kolejno odrzucone i usunięte z kolejki. Naszym ostatecznym rozwiązaniem jest to z kosztem ścieżki 132.

Może się jednak zdarzyć, że należy powrócić do którego z wcześniejszych węzłów i je rozwijać, do momentu, w którym dolne ograniczenie przekroczy najlepsze rozwiązanie dotychczas, bądź też otrzymamy lepsze rozwiązanie problemu od tego dotychczas. Sposób rozgałęziania pozostaje analogiczny.

3. Opis eksperymentu

3.1 Założenia

Pomiary czasu działania algorytmów w zależności od rozmiaru problemu wykonane zostały dla 7 różnych reprezentatywnych wartości N (N – liczba wierzchołków grafu). Dla każdej wartości N wygenerowanych zostało po 100 losowych instancji problemu, dla których przetestowany został algorytm. Wyniki pomiarów czasu przedstawione w poniższym sprawozdaniu zostały uśrednione spośród tych 100 przebiegów algorytmu dla danej liczby wierzchołków N.

3.2 Sposób generowania grafów oraz pomiar czasu

Generator liczb losowych stanowiła struktura *generator,* wykorzystująca maszynę losującą: 32-bit Mersenne twister engine z biblioteki random, czyli *std::mt19937 gen.*

Listing 1. Struktura zawierająca metody służące do inicjalizacji generatora oraz wygenerowania liczby pseudolosowej z podanego zakresu.

```
struct generator {
    std::uniform_int_distribution<> dist;
    std::random_device rd;
    std::mt19937 gen;

    void setGen() {
        gen = std::mt19937(rd());
    }

    void setDist(int newLimit) {
        dist = std::uniform_int_distribution<>(1, newLimit);
    }

    int getRandomNum() {
        return dist(gen);
    }
};
```

Jako, że dla eksperymentu należało przyjąć grafy pełne i asymetryczne, w celu generowania grafu reprezentowanego przez macierz sąsiedztwa, napisana została prosta pętla wstawiająca jako wagi krawędzi losowo wygenerowane liczby z zadanego przedziału – dla tego eksperymentu były to wagi od 1 do 1000000. Na przekątnej grafu wstawiana była natomiast wartość -1, oznaczająca brak połączenia wierzchołka z samym sobą.

Listing 2. Metoda generująca losową instancję grafu w postaci macierzy sąsiedztwa.

```
// Metoda generuje macierz dla asymetrycznego grafu pełnego o zadanej
liczbie wierzcholkow
void Matrix::generate(int newSize, RandomDataGenerator::generator
*numberGenerator) {
    clearData();
    numberGenerator->setDist(distanceValueLimit);
    size = newSize;

    matrix = new int *[size];
    for (int i = 0; i < size; i++) {
        matrix[i] = new int[size];
    }
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (i == j) {
                matrix[i][j] = -1;
            } else {
                matrix[i][j] = numberGenerator->getRandomNum();
            }
        }
    }
    exists = true;
    if (!testing) {
        displayMatrix();
    }
}
```

W celu uzyskania dużej dokładności pomiarowej czasu (o rozdzielczości mikrosekundowej), wykorzystana została funkcja *QueryPerformanceCounter().*

```
long long int Timer::read_QPC() {
    LARGE_INTEGER count;
    QueryPerformanceCounter( lpPerformanceCount: &count);
    return ((long long int) count.QuadPart);
}
```

W ten sposób zmierzony został stan licznika przed rozpoczęciem testowanej operacji oraz po jej wykonaniu. Następnie, Końcowy rezultat uzyskuje się przez podzielenie odmierzonej liczby impulsów licznika przez częstotliwość, którą otrzymuje się za pomocą wywołania *QueryPerformanceFrequency().*

```
double Timer::getMicroSecondsElapsed() {
    long long int elapsed;
    QueryPerformanceFrequency( lpFrequency: (LARGE_INTEGER *) &frequencyPerf);
    elapsed = timerStop - timerStart;
    return ((1000000.0 * elapsed) / frequencyPerf);
}
```

3.3 Ogólny plan przeprowadzania eksperymentu

- Wszystkie algorytmy zostały przetestowane dla 100 losowo wygenerowanych instancji grafu o zadanej liczby wierzchołków – N. Polegało to na tym, że kolejno generowano losową instancję grafu, następnie uruchomiany został dany algorytm dla tego grafu. Za graniczną wartość czasową, po przekroczeniu której algorytm został przerywany – przyjęto czas 2 minut = 120 000 milisekund. Gdy algorytm przekroczył ten limit, był on przerywany, a jego wynik nie był uwzględniany do wyliczania średniej. Następnie następowało wygenerowanie nowej, losowej instancji grafu oraz kolejny przebieg pętli.
- Poniżej przedstawione wyniki pomiarów to wartości uśrednione na podstawie 100 przebiegów danego algorytmu dla losowo wygenerowanych instancji grafów.
- Wartości pomiarów czasu przedstawione zostały w jednostce Milisekund (ms).

4. Zaimplementowane algorytmy

4.1 Brute Force

4.1.1 Opis implementacji

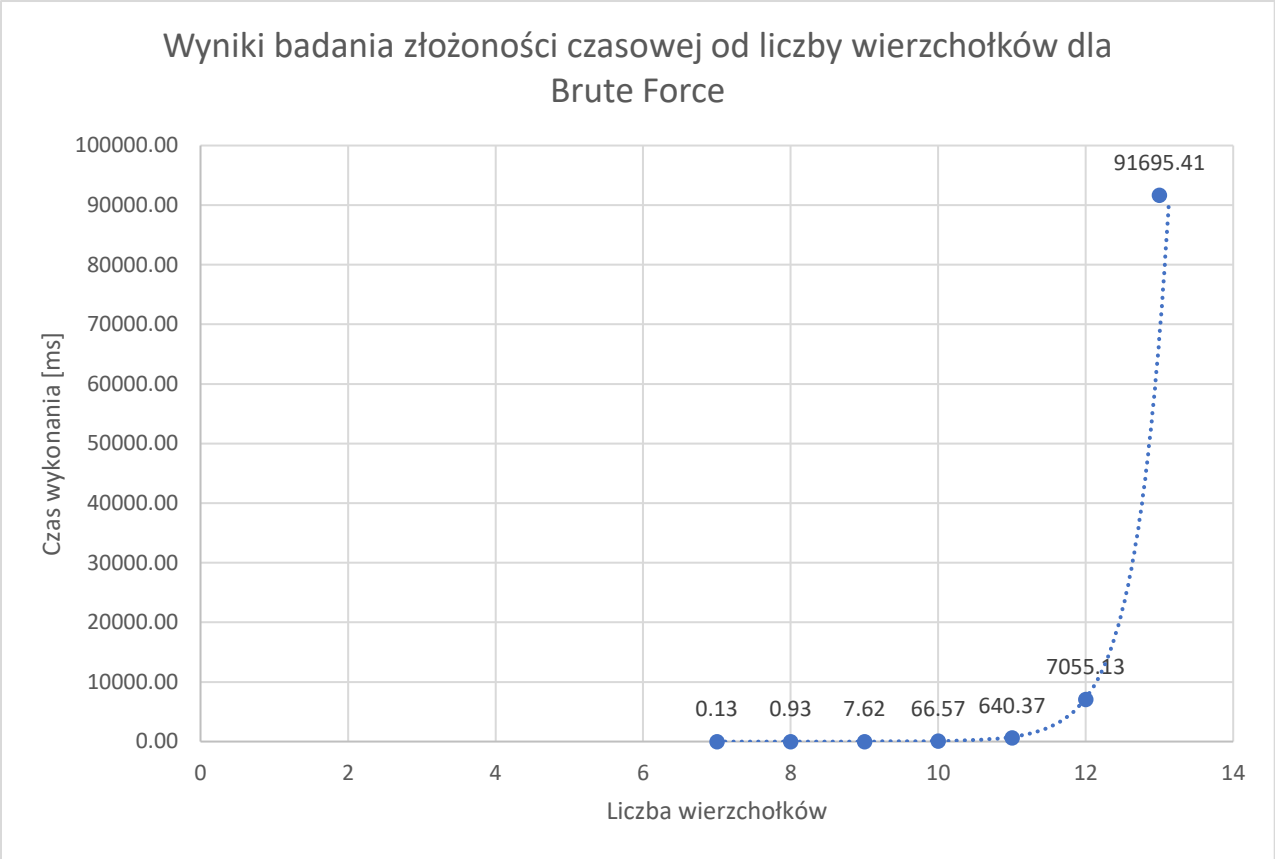
Algorytm przeglądu zupełnego został zaimplementowany bez użycia dodatkowych bibliotek, w wariancie rekurencyjnym – zaczynając od wierzchołka startowego, w pętli sprawdzane są rekurencyjnie wszystkie możliwe kombinacje ścieżki Komiwojażera. Suma wag krawędzi aktualnej ścieżki zapamiętana jest w postaci sumy pomocniczej, natomiast w celu zapamiętania ścieżki wykorzystana została prosta implementacja listy, napisana na wzór tej zaimplementowanej na projekt „Struktury Danych i Złożoność Obliczeniowa” z uprzednich semestrów. Kolejne wierzchołki są dodawane na koniec listy w miarę odwiedzania kolejnych wierzchołków dla danej permutacji lub usuwane z jej końca podczas „cofania się” w rekurencji. W przypadku gdy ścieżka zawiera wszystkie możliwe wierzchołki, do sumy dodawana jest waga krawędzi z ostatniego odwiedzonego wierzchołka do wierzchołka startowego. Gdy koszt ścieżki jest najlepszy dotychczas, suma pomocnicza wag krawędzi oraz ścieżka zostają kopiowane jako wynik. Schemat ten jest powtarzany aż do sprawdzenia wszystkich permutacji ścieżki. Wybór wierzchołka startowego nie ma tutaj znaczenia.

4.1.2 Wyniki pomiarów

Poniższa tabela przedstawia uśrednione wyniki czasu wykonywania algorytmu:

Przegląd zupełny (Brute Force)							
Liczba wierzchołków	7	8	9	10	11	12	13
Czas wykonywania [ms]	0.13	0.93	7.62	66.57	640.37	7055.13	91695.41

Tabela 1. Wyniki eksperymentu dla algorytmu przeglądu zupełnego.



Wykres 1. Wykres przedstawiający wyniki pomiarów czasu dla algorytmu przeglądu zupełnego w zależności od liczby wierzchołków.

Badania zakończono dla grafów o liczbie wierzchołków równej 13, ponieważ problem dla grafów o wymiarze 14 wierzchołków wykonywał się powyżej zakładanego czasu granicznego 2 minut.

4.2 Branch and Bound (algorytm Little'a)

4.2.1 Opis implementacji

Pierwszym krokiem mojej implementacji jest pobranie macierzy wejściowej oraz jej zredukowanie po wierszach i kolumnach. Suma kosztów redukcji stanowi dolne ograniczenie pierwszego węzła drzewa – „korzenia". Węzeł ze zredukowaną macierzą wejściową opakowywany jest w obiekt ***BranchAndBoundNode***, który następnie trafia do kolejki priorytetowej.

- W przypadku mojej implementacji tego algorytmu, poddrzewa reprezentowane są przez obiekt zawierający stan macierzy dla danego poddrzewa, listę wybranych krawędzi oraz listę pod-ścieżek, jakie tworzą wybrane krawędzie poddrzewa (wykorzystywana do zablokowania wyboru krawędzi, która powodowałaby powstanie cyklu innego niż cykl Hamiltona – w tym algorytmie należy wybierać taką krawędź, która powoduje powstanie możliwie najdłuższej ścieżki).

Strukturą wykorzystywaną w celu przechowywania zakolejkowanych węzłów drzewa stanowi ***std::priority_queue*** z biblioteki STL, ze względu na złożoność wstawiania nowego elementu która wynosi $O(log\ n)$. Węzły są w niej sortowane nierosnąco, według wartości ich dolnego ograniczenia.

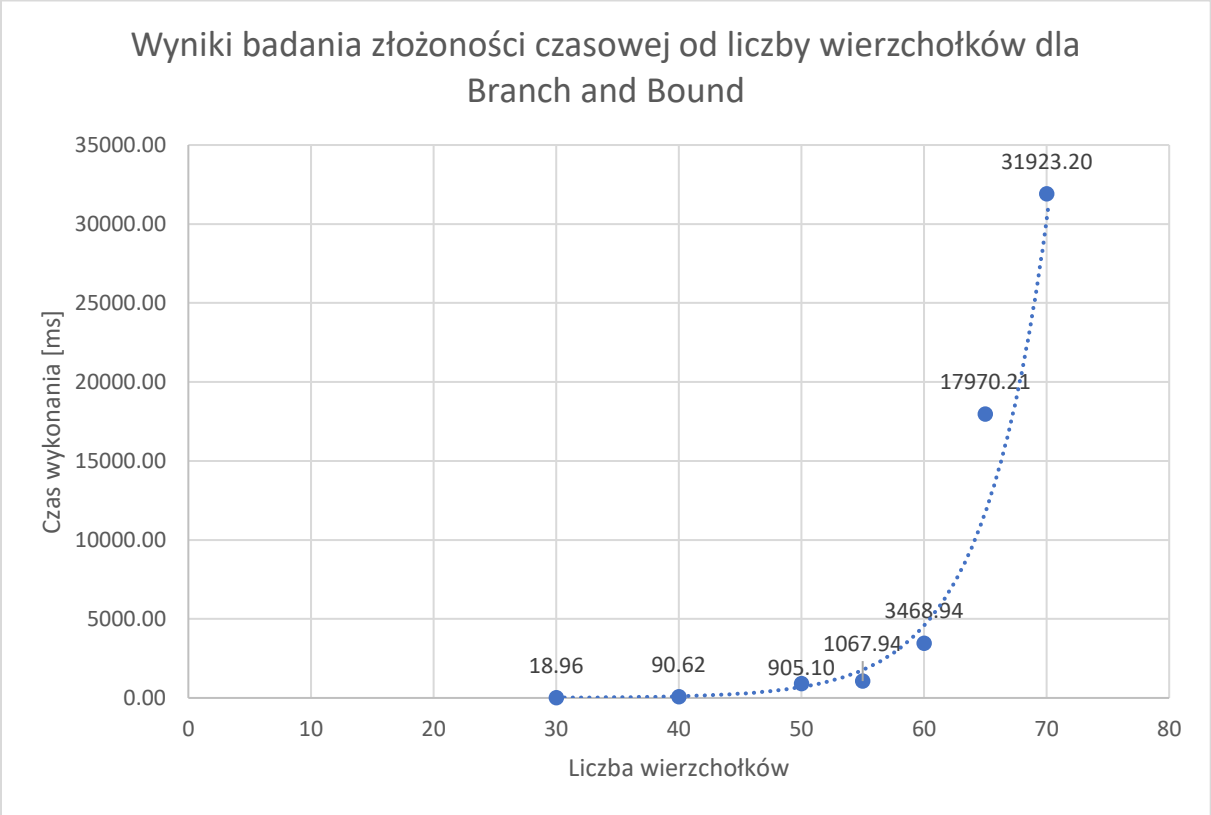
W głównej pętli pobieramy węzeł z kolejki priorytetowej i go z niej usuwamy. Następnie wybierana jest krawędź w postaci pary (wiersz, kolumna), po czym dokonywany jest podział na dwa poddrzewa, zgodnie z zasadą działania algorytmu Little'a – lewe, zawierające krawędź oraz prawe, z ową krawędzią jako odrzuconą. Po podziale na poddrzewa względem wybranej krawędzi, gdy zostało wybranych więcej krawędzi i tworzą one pod-ścieżkę – należy zablokować powstawanie cykli nie będących cyklem Hamiltona przez wpisanie wartości **INF** (INT_MAX) pod indeksem macierzy reprezentującej krawędź powracającą do początkowego wierzchołka pod-ścieżki. Pod-ścieżki mogą być oczywiście tworzone przez 2 bądź więcej krawędzi. Po aktualizacji obu macierzy reprezentujących pod-drzewa, dokonujemy ponownej redukcji obu macierzy oraz aktualizujemy dolne ograniczenie o koszt redukcji. Oba węzły trafiają następnie do kolejki, po czym wracamy do początku pętli. Szczególnym przypadkiem jest, gdy węzeł został zredukowany do macierzy o wymiarach 2x2. W tym wypadku dobieramy ostatnie dwie krawędzie, tak aby wszystkie krawędzie tworzyły cykl Hamiltona. W ten sposób otrzymujemy rozwiązanie. Algorytm jednak kontynuuje działanie, dopóki nie sprawdzimy pozostałych węzłów drzewa w kolejce, czy nie istnieje węzeł o mniejszym dolnym ograniczeniu niż aktualne rozwiązanie. W takim wypadku kontynuujemy rozgałęzianie na pod-drzewa.

4.2.2 Wyniki pomiarów

Poniższa tabela przedstawia uśrednione wyniki czasu wykonywania algorytmu Branch and Bound. Analizując wyniki dla tego algorytmu należy brać pod uwagę informację o ilości jego przerwanych wywołań, znajdującej się na następnej stronie.

Branch and Bound							
Liczba wierzchołków	30	40	50	55	60	65	70
Czas wykonywania [ms]	18.96	90.62	905.10	1067.94	3468.94	17970.21	31923.20

Tabela 2. Wyniki eksperymentu dla algorytmu Branch and Bound Little'a.

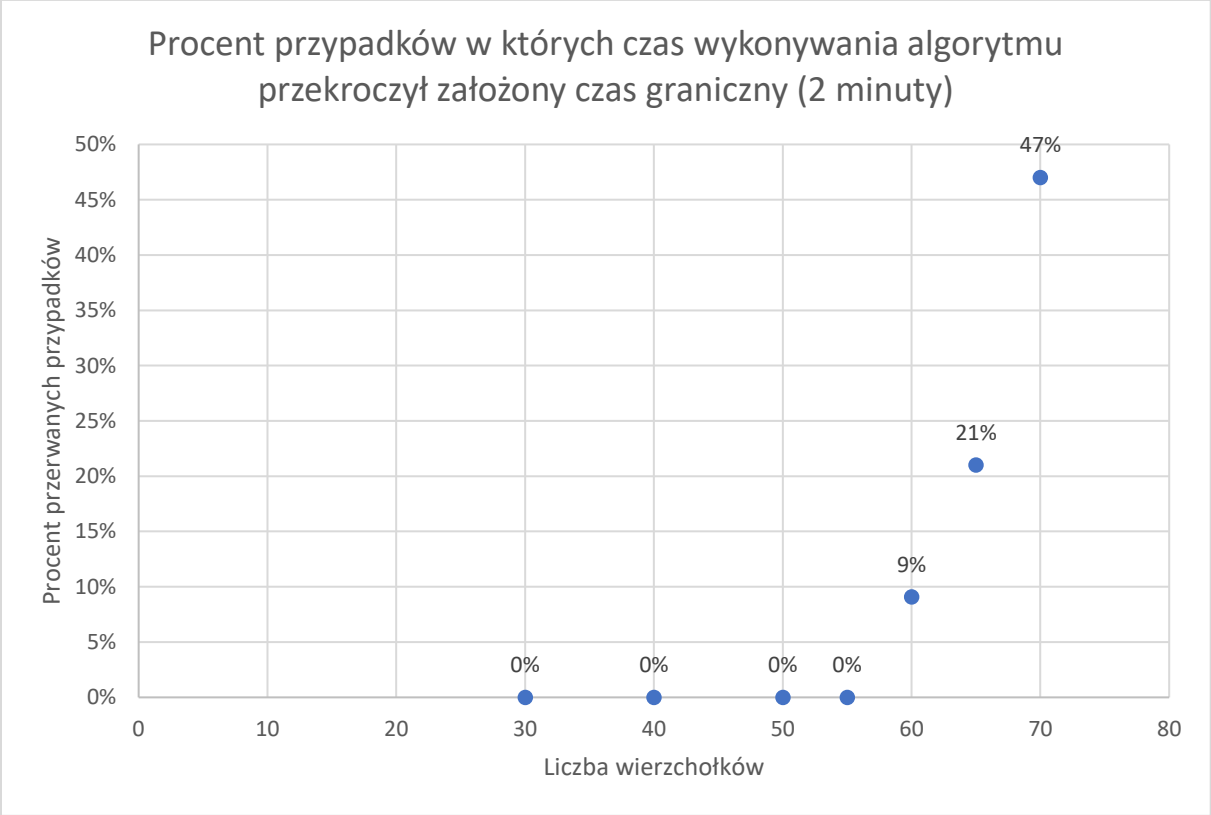


Wykres 2. . Wykres przedstawiający wyniki pomiarów czasu dla algorytmu Little'a w zależności od liczby wierzchołków.

Poniższa tabela obrazuje ilość (w procentach) przerwanych wywołań algorytmu Branch and Bound dla losowej instancji problemu o rozmiarze N wierzchołków. Za czas graniczny przyjęto 2 minuty, po przekroczeniu którego algorytm był przerywany i generowano nową instancję problemu o danym rozmiarze.

Branch and Bound							
Liczba wierzchołków	30	40	50	55	60	65	70
Procent przypadków, w których czas wykonywania przekroczył zakładany czas graniczny (2 minuty)	0%	0%	0%	0%	9%	21%	47%

Tabela 3. Procent przerwanych wywołań algorytmu Little'a z powodu przekroczenia czasu granicznego w zależności od liczby wierzchołków.



Wykres 3. Wykres przedstawiający procent przerwanych wywołań algorytmu Little'a w zależności od wielkości instancji.

Dalsze pomiary, dla większej ilości wierzchołków okazały się niemożliwe do wykonania, gdyż zużycie pamięci było zbyt duże.

4.2.3 Opis sposobu obliczania ograniczenia oraz wyboru krawędzi

Dla algorytmu Little'a dolne ograniczenie to suma ograniczenia węzła rodzica oraz wartości redukcji macierzy zaktualizowanej o wybraną krawędź (macierz poddrzewa zawierającego wybraną krawędź w przypadku lewego poddrzewa, bądź macierz „bez" wybranej krawędzi w przypadku prawego poddrzewa), lub jedynie suma wartości redukcji w przypadku pierwszego węzła - „korzenia"). W mojej implementacji za redukcję względem wierszy, redukcję względem kolumn oraz wybór krawędzi odpowiadają osobne funkcje, które przedstawione zostały poniżej.

4.2.3.1 Redukcja macierzy względem wierszy

Listing 3. Metoda redukująca kolejno wiersze macierzy, zwracająca sumę wartości redukcji.

```
// Funckja sluzaca do redukcji wierszy macierzy
int BranchAndBound::reduceRows(int **matrix, int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        int min = INT_MAX;
        for (int j = 0; j < size; j++) {
            if (i == j) {
                continue;
            }
            if (min > matrix[i][j]) {
                min = matrix[i][j];
            }
        }
        if (min != INT_MAX) {
            sum += min;
        }
        if (min == 0 || min == INT_MAX) {
            continue;
        }
        for (int j = 0; j < size; j++) {
            if (i == j || matrix[i][j] == INF || matrix[i][j] == 0) {
                continue;
            }
            matrix[i][j] -= min;
        }
    }
    return sum;
}
```

W tej funkcji iterujemy kolejno po wszystkich wierszach macierzy, szukając minimalnej wartości we wierszu – nie będącej zerem lub INF (za wartość INF przyjęto INT_MAX, wartość maksymalną dla typu Integer). Jeśli najmniejsza wartość w wierszu wynosi 0, to nie redukujemy wiersza i przechodzimy do następnego wiersza. W przeciwnym wypadku iterujemy jeszcze raz przez wszystkie elementy aktualnego wiersza, odejmując jednocześnie od nich wartość minimalną. W ten sposób otrzymujemy w każdym wierszu co najmniej jedno 0, które stanowi w dalszej części algorytmu „kandydata na wybór krawędzi" i dokonania podziału.

4.2.3.2 Redukcja macierzy względem kolumn

Listing 4. Metoda redukująca kolejno kolumny macierzy, zwracająca sumę wartości redukcji.

```
// Funckja sluzaca do redukcji kolumn macierzy
int BranchAndBound::reduceColumns(int **matrix, int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        int min = INT_MAX;
        for (int j = 0; j < size; j++) {
            if (i == j || matrix[j][i] == INF) {
                continue;
            }
            if (min > matrix[j][i]) {
                min = matrix[j][i];
            }
        }
        if (min != INT_MAX) {
            sum += min;
        }
        if (min == 0 || min == INT_MAX) {
            continue;
        }
        for (int j = 0; j < size; j++) {
            if (i == j || matrix[j][i] == INF || matrix[j][i] == 0) {
                continue;
            }
            matrix[j][i] -= min;
        }
    }
    return sum;
}
```

Funkcja została napisana w sposób analogiczny do funkcji służącej redukcji wierszy, jednak w tym wypadku iterujemy kolejno po wszystkich kolumnach macierzy, szukając minimalnej wartości w danym wierszu. Jeśli wartość minimalna w danej kolumnie nie jest zerem, iterujemy przez daną kolumnę jeszcze raz, odejmując tym samym wartość minimalną w danej kolumnie od wszystkich elementów w tej kolumnie.

Po wywołaniu kolejno funkcji ***reduceRows()*** oraz ***reduceColumns()***, w każdym wierszu (który nie został już wcześniej wykreślony poprzez wpisanie *INT_MAX* z powodu wyboru krawędzi) oraz w każdej kolumnie (która nie została już wcześniej wykreślona poprzez wpisanie *INT_MAX* z powodu wyboru krawędzi) co najmniej jeden element będzie posiadał wartość równą 0, które są rozpatrywane w dalszej części algorytmu, podczas wyszukiwania krawędzi, która powoduje największy wzrost dolnego ograniczenia dla rozwiązań niezawierających tej krawędzi.

Dolne ograniczenie poddrzewa, które reprezentowane jest przez zredukowaną macierz, będzie stanowić suma wartości redukcji wierszy – zwróconą przez funkcję ***reduceRows()*** oraz wartości redukcji kolumn – zwróconą przez funkcję ***reduceColumns()***.

4.2.3.3 Wybór krawędzi

Listing 5. Funkcje służące do wyboru łuku, po którym dokonywany pędzie podział drzewa.

```
// Funkcja sluzaca do wyboru luku, ktory powoduje najwiekszy wzrost dolnego
ograniczenia dla rozwiazan niezawierajacych tego luku
std::pair<int, std::pair<int, int>>
BranchAndBound::chooseBestCaseEdge(int **matrix, int size, bool &outSuccess) {
    int x = 0, y = 0;
    int value = INT_MIN;
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (i == j) continue;
            if (matrix[i][j] == 0) {
                int minimum = getMinimumDefined(matrix, i, j, size);
                if (minimum > value) {
                    value = minimum;
                    x = i;
                    y = j;
                    outSuccess = true;
                }
            }
        }
    }
    return std::make_pair(value, std::make_pair(x, y));
}

// Funkcja pomocnicza sluzaca do wyboru kandydatow na wybor luku
int BranchAndBound::getMinimumDefined(int **matrix, int row, int column, int
size) {
    int valueRow = INT_MAX;
    int valueColumn = INT_MAX;
    for (int i = 0; i < size; i++) {
        if (i != column && matrix[row][i] != INF && matrix[row][i] < valueRow)
{
            valueRow = matrix[row][i];
        }
        if (i != row && matrix[i][column] != INF && matrix[i][column] <
valueColumn) {
            valueColumn = matrix[i][column];
        }
    }
    return valueRow + valueColumn;
}
```

Po dokonaniu redukcji macierzy, wywoływana jest funkcja ***chooseBestCaseEdge()*** służąca do wyboru krawędzi, po której dokonywany będzie podział na dwa poddrzewa (krawędź ta powoduje największy wzrost dolnego ograniczenia dla rozwiązań niezawierających tej krawędzi). W tej funkcji iterujemy kolejno przez wszystkie elementy macierzy, w poszukiwaniu zer. Gdy znalezione zostaje 0, wywoływana jest funkcja ***getMinimumDefined***() – jest to funkcja pomocnicza, wyliczająca wzrost dolnego ograniczenia, które spowoduje wybór tej krawędzi na i dokonanie względem niej podziału na kolejne poddrzewa. W celu wyznaczenia tej wartości, w funkcji ***getMinimumDefined***() iterujemy jednocześnie po wierszu oraz kolumnie (reprezentujących aktualnie wybraną krawędź względem której możliwy jest podział) – wyznaczając najmniejszą wartość w danym wierszu oraz najmniejszą wartość w danej kolumnie (nie będącymi pod indeksami *macierz[wiersz][kolumna]* – wartość macierzy pod indeksami wybranej krawędzi zawsze wynosi 0).

Efektem działania funkcji ***chooseBestCaseEdge***() będzie zwrócenie wybranej krawędzi oraz wzrost dolnego ograniczenia po jej wyborze.

5. Wnioski

Wyniki eksperymentu pokazują, że metoda przeglądu zupełnego (algorytm Brute Force) staje się kompletnie niepraktyczny już przy kilkunastu wierzchołkach instancji grafu. Jak można zaobserwować na wykresie, czas wykonywania się algorytmu wzrasta drastycznie, gdy liczba wierzchołków przekroczyła 12 – widzimy największy przyrost czasu względem poprzednich rozmiarów.

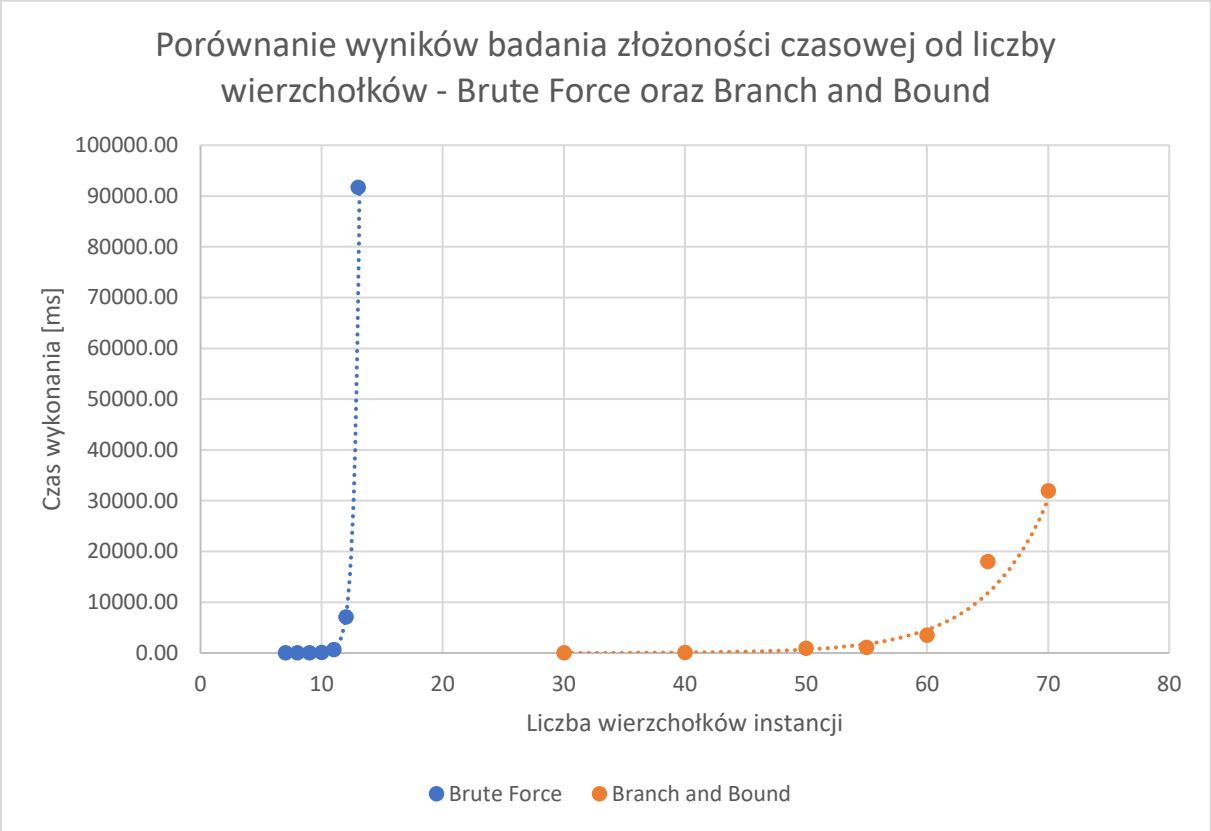
Algorytm Branch and Bound (Little'a) jest bardzo efektywny, aż do osiągnięcia rozmiaru instancji problemu o rozmiarze około 55 wierzchołków. Wtedy czas wykonania algorytmu zaczyna zauważalnie rosnąć. Przy 60 wierzchołkach zdarzyły się już przypadku, w których czas działania algorytmu przekroczył założony czas graniczny 2 minut – przez co jego działanie zostało zakończone (9% ze 100). Przy 65 wierzchołkach czas wykonywania algorytmu wzrósł już znacząco, a ilość przerwanych wywołań algorytmu wyniosła już 21%. Przy rozmiarze instancji na poziomie 70 wierzchołków algorytm zdecydowanie staje się niepraktyczny, ponieważ prawie połowa przypadków wywołania algorytmu została przerwanych (47%) z powodu przekroczenia dopuszczalnych 2 minut czasu granicznego. Dodatkowo zużycie pamięci przy tym rozmiarze instancji potrafi osiągnąć bardzo wysoki poziom. Zauważyć także było można, że wyniki czasu wykonania w przypadku instancji o rozmiarze 65 oraz 70 były bardzo zróżnicowane – od kilkunastu sekund do nawet lekko poniżej 2 minut.

Przy analizie wykresu dla algorytmu Branch and Bound należy zatem brać pod uwagę procent przerwanych wywołań algorytmu, gdyż świadczy to o niespójnej złożoności obliczeniowej dla tego rozwiązania w zależności od wygenerowanej instancji grafu.

Być może przyczyną dużego zużycia pamięci w przypadku części wywołań algorytmu Little'a dla losowej instancji stanowi fakt, że na rzecz mojej implementacji tego algorytmu przyjęto strategię wyboru kolejnych węzłów według minimalnej wartości dolnego ograniczenia. Strategia ma zarówno swoje wady, jak i zalety. Ogromną zaletą jest to, że doprowadza najszybciej do optymalnego rozwiązania problemu. Możliwe jednak, że przyjęcie strategii podziału „z kolejnego otrzymanego węzła" okazałoby się lepsze w niektórych wypadkach, ponieważ najszybciej doprowadza ona do otrzymania wyniku (cyklu Hamiltona), co pozwoliłoby na odrzucenie części węzłów przechowywanych w kolejce priorytetowej, których dolne ograniczenie jest większe i przyczyniłoby się na mniejsze zużycie pamięci podczas wykonywania algorytmu.

5.1 Porównanie wyników eksperymentu dla obu algorytmów

Poniżej przedstawiony został wykres porównujący wyniki eksperymentu dla obu algorytmów.



Wykres 4. Wykres przedstawiający wyniki pomiarów czasu dla algorytmu Brute Force oraz Branch and Bound (Little'a).

Należy wziąć poprawkę na fakt, iż wykres dla algorytmu Branch and Bound nie uwzględnia czasów przerwanych wywołań dla losowo wygenerowanej instancji, których czas wykonywania przekroczył 2 minuty. Ostateczny czas wykonania w takim wypadku mógłby wynieść czas zbliżony jak w przypadku Brute Force dla danego rozmiaru problemu, szacując w sposób najbardziej pesymistyczny.

6. Literatura

6.1 AN ALGORITHM FOR THE TRAVELING SALESMAN PROBLEM – John D. C. Little, Katta G. Murty, Dura W. Sweeney and Caroline Karel

https://www.jstor.org/stable/pdf/167836.pdf?casa_token=dwFpExZ_AxYAAAAA:k1GTMzAx8PS5JH8RiichVEgDEk_OU0n_EUjwxRQFJ3DDl3mPJ8y6GQWxVGt1P25oLxroICbGF7v0J5CXKQdHJjkznIWY9Sn7PNVlInckDYnGHXbq

https://dspace.mit.edu/bitstream/handle/1721.1/46828/algorithmfortrav00litt.pdf?sequence=1&isAllowed=y

6.2 Metoda podziału i ograniczeń – Paweł Zieliński

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

6.3 Problem TSP – Wikipedia

https://en.wikipedia.org/wiki/Travelling_salesman_problem

6.4 Problem Komiwojażera – Liceum w Tarnowie

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

6.5 Branch and Bound—Why Does It Work?

https://rjlipton.wpcomstaging.com/2012/12/19/branch-and-bound-why-does-it-work/