Politechnika Wrocławska

Wydział Elektroniki
Katedra Informatyki Technicznej

PROJEKTOWANIE EFEKTYWNYCH ALGORYTMÓW
PROJEKT

# Projekt nr 1 - Implementacja i analiza efektywności algorytmu podziału i ograniczeń i programowania dynamicznego.

Miron Oskroba, 236705

Prowadzący - dr inż. Jarosław Mierzwa
Termin zajęć: Pn $15^{15}$

Wrocław, 14.11.2020

# Spis treści

# 1 Wstęp teoretyczny

Zadaniem do wykonania jest implementacja oraz dokonanie analizy wybranych algorytmów dla asymetrycznego problemu komiwojażera:

1. **Brute Force Algorithm- z ang. "Algorytm Przeglądu Zupełnego"**
2. **Branch & Bound Algorithm- z ang. "Algorytm Podziału i ograniczeń"**

## 1.1 Opis rozpatrywanego problemu

Rozpatrywanym problemem jest implementacja asymetrycznego problemu komiwojażera (ang. "asymmetric travelling salesman problem" - **ATSP**). Jest to problem polegający na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym, w zoptymalizowany sposób. Nazwa pochodzi od zobrazowania problemu, przedstawionego z punktu widzenia wędrownego sprzedawcy, którego celem jest odwiedzenie pewnej skończonej ilości miast, mając dane odległości między nimi. Asymetryczny problem różni się od symetrycznego, ponieważ nie jest powiedziane, że droga z przykładowego miasta A do miasta B jest taka sama jak z miasta B do miasta A. Celem komiwojażera jest odwiedzenie każdego miasta raz najmniejszym możliwym kosztem (najkrótszą drogą), tak aby jego droga zaczynała się i kończyła w początkowo określonym mieście [1].

## 1.2 Opis algorytmów

Wybrane algorytmy są zoptymalizowane w różnym stopniu. Poniżej znajduje się charakterystyka wybranych algorytmów w zadanym problemie.

1. **Brute Force**
   - Opis algorytmu
     Algorytm przeglądu zupełnego polega na sprawdzeniu wszystkich możliwych dróg z jednoczesnym sprawdzaniem czy aktualna droga jest najlepsza - następuje wtedy aktualizacja wyniku w problemie**ATSP**. Algorytm sprawdza wszystkie permutacje dla wektora zadanej długości - w zależności od danych wejściowych.

   - Szacowanie złożoności obliczeniowej
     Metoda ta jest nieefektywna obliczeniowo ze względu na wykonywaną liczbę iteracji oraz brak logiki do sprawdzania czy warto jest wykonywać daną iterację - iteracja jest po prostu wykonywana. Dlatego wraz ze wzrostem ilości danych wejściowych do przetworzenia czas wykonywania jest znacznie wydłużony. W zaimplementowanym algorytmie dla 'n' miast należy sprawdzić każdą permutację, a takich permutacji jest $n * (n-1)(n-2)... * 1 = n!$. Otrzymujemy więc złożoność rzędu:

     $$\theta(n!)$$

     Co czyni algorytm bezużytecznym czasowo dla większych zbiorów danych [2].

2. **Branch & Bound**
   - Opis algorytmu
     Algorytm podziału i ograniczeń służy do rozwiązywnia problemów optymalizacyjnych. Różne wersje algorytmu charakteryzują się trudnością i sposobem implementacji. W przypadku problemu **ATSP** algportym zostanie użyty w celu zminimalizowania liczby odwiedzanych wierzchołków przez zawarcie odpowiedniej logiki, co pozwoli na zmniejszenie jego złożoności obliczeniowej i co za tym idzie - krótszy

czas wykonywania algorytmu. Przeszukiwanie wszerz zilustrować można jako drzewo przestrzeni stanów - realizujące wszystkie możliwe ścieżki, którymi mógłby pójść komiwojażer. Optymalizacja logiczna polega na liczeniu potencjału poddrzewa danego wierzchołka uwzględniając występujące w nim różne ścieżki kosztów oraz sprawdzenie czy tak wyznaczona granica jest lepsza od aktualnego najlepszego rozwiązania. Taka implementacja pozwala na znacznie szybsze wykonywanie, ponieważ gdy tylko algorytm zauważy, że nie warto dalej sprawdzać - przechodzi do sprawdzania następnego wierzchołkay. Algorytm został zaprogramowany w sposób rekursywny.

- Szacowanie złożoności obliczeniowej
  W najgorszym przypadku może się zdarzyć, że pomimo ułatwień logicznych dane wejściowe skonstruowane będą w taki sposób, że złożoność obliczeniowa będzie dokładnie taka sama jak w przypadku Brute Force, czyli $\theta(n!)$. Szacowanie złożoności obliczeniowej dla tego algorytmu nie należy do najprostszych zadań. Dokładna analiza złożoności obliczeniowej została przedstawiona w [3] np. na str. 8, wg której średnia złożoność algorytmu podziału i ograniczeń wynosi:

$$\theta(n^3 ln^2(n))$$

# 2 Przykłady praktyczne

1. **Brute Force**
   Z wytycznych projektowych: "...dla przeglądu zupełnego nie robimy przykładu".
2. **Branch & Bound**
   Algorytm krok po kroku dla losowo wygenerowanych danych, N = 3 (ilość miast). Czyli pełne wykonanie funkcji *void find_best path();* z klasy źródłowej *BranchAndBound*. Plik nagłówkowy **BranchAndBound.h**:

```cpp
//
// Created by Aron on 2020-11-09.
//
#ifndef CODE_BRANCHANDBOUND_H
#define CODE_BRANCHANDBOUND_H

#include <vector>

class BranchAndBound {
private:
    std::vector<std::vector<int>> cost_matrix;//input data
    std::vector<int> path_best;
    std::vector<int> visited;//vertices visited: 0=not visited, 1=visited
    int amount_of_vertices;
    int cost_best;

private:
    int first_min(int vertex);//gets first minimum of a vertex
    int second_min(int vertex);//gets second minimum of a vertex
    void update_path_best(std::vector<int> path_curr);//updates result

public:
    explicit BranchAndBound(std::vector<std::vector<int>> matrix);//constructor
    ~BranchAndBound();//destructor
```

```

        void find_best_path();//entry point of an algorithm
        void ATSP(std::vector<int> path_curr, int cost_curr, int bound_lower_curr, int
        ↪ level);//recursive algorithm
        void print_best_path();
    };
    #endif //CODE_BRANCHANDBOUND_H
```

- **Przebieg algorytmu praktycznego dla danych wejściowych:**

| | | |
|---|---|---|
| 3 | | |
| 0 | 25 | 14 |
| 59 | 0 | 61 |
| 46 | 66 | 0 |

Tablica 1: Dane wejściowe dla praktycznej realizacji algorytmu B&B

```
0: Wyczyszczenie i zainicjowanie wektorow visited & path_best
1: Wyznaczenie bound_lower_curr = 135
2: Aktualizacja visited[0] = 1 & path_curr[0] = 0
3: Wywolano ATSP(path_curr = {0,-1,-1,-1}, cost_curr = 0, bound_lower_curr = 135, level =
↪ 1);
4: vertex_before = 0
5: Sprawdz, czy amount_of_vertices =?= level -> 0
6: vertex_curr = 0
7: Sprawdz, czy aktualny wierzcholek byl juz sprawdzany -> 0
8: vertex_curr = 1
9: Sprawdz, czy aktualny wierzcholek byl juz sprawdzany -> 1
10: Aktualizacja kosztu aktualnego przejscia, cost_curr =  25
11: Aktualizacja bound_lower_curr w zaleznosci od level: bound_lower_curr = 99
12: Sprawdz, czy warto zglebiac aktualny wierzcholek? -> 1
13: visited[1] = 1;
14: path_curr[1] = 1;
15: Wywolaj ATSP dla nastepnego poziomu.
16: Wywolano ATSP(path_curr = {0,1,-1,-1}, cost_curr = 25, bound_lower_curr = 99, level =
↪ 2);
17: vertex_before = 1
18: Sprawdz, czy amount_of_vertices =?= level ->        0
19: vertex_curr = 0
20: Sprawdz, czy aktualny wierzcholek byl juz sprawdzany -> 0
21: vertex_curr = 1
22: Sprawdz, czy aktualny wierzcholek byl juz sprawdzany -> 0
23: vertex_curr = 2
24: Sprawdz, czy aktualny wierzcholek byl juz sprawdzany -> 1
25: Aktualizacja kosztu aktualnego przejscia, cost_curr =  86
26: Aktualizacja bound_lower_curr w zaleznosci od level: bound_lower_curr = 46
27: Sprawdz, czy warto zglebiac aktualny wierzcholek? -> 1
28: visited[2] = 1;
29: path_curr[2] = 2;
```

```
30: Wywolaj ATSP dla nastepnego poziomu.
31: Wywolano ATSP(path_curr = {0,1,2,-1}, cost_curr = 86, bound_lower_curr = 46, level =
    3);
32: vertex_before = 2
33: Sprawdz, czy amount_of_vertices =?= level ->        1
34: cost_total = 132
35: Sprawdz, czy cost_total ?<? cost_best ->    1
36: cost_total = 132
37: Aktualizacja najlepszej sciezki: path_best= {0,1,2}
38: Przywroc zmiany:
39:  cost_curr = 25
40:  bound_lower_curr = 99
41: Aktualizacja odwiedzonych wierzolkow
42: Przywroc zmiany:
43:  cost_curr = 0
44:  bound_lower_curr = 135
45: Aktualizacja odwiedzonych wierzolkow
46: vertex_curr = 2
47: Sprawdz, czy aktualny wierzcholek byl juz sprawdzany -> 1
48: Aktualizacja kosztu aktualnego przejscia, cost_curr =  14
49: Aktualizacja bound_lower_curr w zaleznosci od level: bound_lower_curr = 105
50: Sprawdz, czy warto zglebiac aktualny wierzcholek? -> 1
51: visited[2] = 1;
52: path_curr[1] = 2;
53: Wywolaj ATSP dla nastepnego poziomu.
54: Wywolano ATSP(path_curr = {0,2,-1,-1}, cost_curr = 14, bound_lower_curr = 105, level
    = 2);
55: vertex_before = 2
56: Sprawdz, czy amount_of_vertices =?= level ->        0
57: vertex_curr = 0
58: Sprawdz, czy aktualny wierzcholek byl juz sprawdzany -> 0
59: vertex_curr = 1
60: Sprawdz, czy aktualny wierzcholek byl juz sprawdzany -> 1
61: Aktualizacja kosztu aktualnego przejscia, cost_curr =  80
62: Aktualizacja bound_lower_curr w zaleznosci od level: bound_lower_curr = 43
63: Sprawdz, czy warto zglebiac aktualny wierzcholek? -> 1
64: visited[1] = 1;
65: path_curr[2] = 1;
66: Wywolaj ATSP dla nastepnego poziomu.
67: Wywolano ATSP(path_curr = {0,2,1,-1}, cost_curr = 80, bound_lower_curr = 43, level =
    3);
68: vertex_before = 1
69: Sprawdz, czy amount_of_vertices =?= level ->        1
70: cost_total = 139
71: Sprawdz, czy cost_total ?<? cost_best ->    0
72: Przywroc zmiany:
73:  cost_curr = 14
74:  bound_lower_curr = 105
75: Aktualizacja odwiedzonych wierzolkow
76: vertex_curr = 2
77: Sprawdz, czy aktualny wierzcholek byl juz sprawdzany -> 0
78: Przywroc zmiany:
```

```
79:  cost_curr = 0
80:  bound_lower_curr = 135
81: Aktualizacja odwiedzonych wierzolkow
82: WYNIK 0 - 1 - 2 - 0,  cost: 132
```

# 3 Opis implementacji algorytmu

Do zaimplementowania algorytmu Branch&Bound użyto klasy *vector*. Dane wejściowe macierzy są przechowywane w wektorze wektorów typu *integer*. Deklaracja macierzy kosztów wygląda następująco:

```
        std::vector<std::vector<int>> cost_matrix;//input data
```

Podczas implementacji algorytmu wzorowano się na artykule naukowym dot. algorytmu Branch&Bound [4]. Obliczanie ograniczenia dolnego dla algorytmu:

- Dla korzenia:
  Dla każdego wierzchołka należy dodać dwa najmniejsze koszty krawędzi. Tak otrzymaną sumę należy podzielić na dwa i zaokrąglić w górę w przypadku powstania liczby ułamkowej.

```
        for (int j = 0; j < amount_of_vertices; j++)
            bound_lower_curr += first_min(j) + second_min(j);
        bound_lower_curr = ceil(bound_lower_curr / 2);
```

- Dla poziomu 1:
  Jeżeli aktualnie sprawdzany wierzchołek jest na poziomie pierwszym sprawdzanej drogi, należy od otrzymanej aktualnej wartości ograniczenia dolnego odjąć sumę podzieloną na dwa dwóch krawędzi o najmniejszym koszcie dla wierzchołka aktualnego i wierzchołka poprzedniego (w tym przypadku - korzenia).

```
        if(level == 1)
            bound_lower_curr -= (first_min(vertex_before) + first_min(vertex_curr))/2;
```

- Dla każdego innego poziomu:
  Dla kolejnych poziomów postępujemy analogicznie jak przy poziomie 1, jednak w przypadku wierzchołka poprzedniego zamiast wziąć krawędź o minimalnym koszcie, uwzględniamy krawędź o drugim minimalnym koszcie. kod1

```
        if(level == 1)
                bound_lower_curr -= (first_min(vertex_before) + first_min(vertex_curr) ) /
                 ↪  2;
            else
                bound_lower_curr -= (second_min(vertex_before) + first_min(vertex_curr) )
                 ↪  / 2;
```

- Dalsze postępowanie algorytmu;
  Celem sprawdzenia, czy aktualny wierzchołek jest obiecujący, należy sprawdzić czy suma aktualnego kosztu przejścia i dolnego ograniczenia jest mniejsza od kosztu aktualnej najlepszej drogi. Jeśli tak jest, to należy zgłębić kolejny wierzchołek.

```
    if(bound_lower_curr + cost_curr < cost_best)
            {
                visited[vertex_curr] = 1;
                path_curr[level] = vertex_curr;

                ATSP(path_curr, cost_curr, bound_lower_curr, level + 1);
            }
```

- Opis algorytmu ATSP zawartego w pliku źródłowym BranchAndBound.cpp. Wszystkie zmienne zdefiniowane są w pliku źródłowym BranchAndBound.h.

1. Wyczyszczenie i zainicjowanie wektorow *visited* i *path_best*
2. Wyznaczenie *bound_lower_curr*
3. Aktualizacja *visited[0] = 1* i p*ath_curr[0] = 0*
4. Wywolanie *ATSP(path_curr, 0, bound_lower_curr, 1);*
5. Nadpisanie *vertex_before = path_curr[level - 1];*
6. Sprawdzenie, czy *amount_of_vertices == level*
7. Jeśli tak, to:
   (a) Nadpisanie *cost_total = cost_curr + cost_matrix[vertex_before][path_curr[0]];*
   (b) Sprawdzenie, czy *cost_total < cost_best*
   (c) Jeśli tak, to
      i. Nadpisanie *cost_best = cost_total*
      ii. Aktualizacja najlepszej drogi: Przypisanie *path_curr do path_best*
   (d) return;
8. Powtarzaj *amount_of_vertices* razy, inkrementując *vertex_curr* raz na pętlę, począwszy od *vertex_curr = 0*
9. Sprawdzenie, czy aktualny wierzchołek był już sprawdzany *visited[vertex_curr] == 0*
10. Jeśli tak, to
   (a) Stworzenie kopii *bound_curr_backup = bound_lower_curr*
   (b) Aktualizacja kosztu aktualnego przejścia: *cost_curr += cost_matrix[vertex_before][vertex_curr];*
   (c) Wyznaczenie dolnej granicy wierzchołka dla danego poziomu zagnieżdżenia jest opisane w sekcji 3 - wyznaczaniu dolnego ograniczenia algorytmu B&B.
   (d) Sprawdzenie opłacalności zgłębienia wierzchołka, również opisane w sekcji 3.
11. Powrót z zagnieżdzonej rekursywnie funkcji ATSP lub aktualny wierzchołek nie jest obiecujący: Przywrócenie zmian, tzn. aktualizacja *cost_curr, bound_lower_curr oraz wektora visited.*

# 4 Plan eksperymentu

- Rozmiar używanych struktur danych:
  Struktury, w której przechowywane są informacje o kosztach przejść pomiędzy poszczególnymi wierzchołkami jest tworzona w zależności od zadanego rozmiaru. Sposoby generowania danych są przedstawione poniżej.
- Sposób generowania danych:
  Zostały zaimplementowane dwa sposoby tworzenia danych wejściowych. Zwracają one wektor wektorów typu *integer*, po czym są ładowane w konstruktorach danych algorytmów.
  - Wczytywanie z pliku:
    Klasa odpowiedzialna: *FileHandler*. Funkcja najpierw pobiera ilośc wierszy, przygotowuje rozmiar struktury dla pobranej liczby, po czym wypełnia strukturę kolejnymi wartościami z pliku .txt. Funkcja prezentuje się następująco:

```cpp
vector<vector<int>> FileHandler::read_from_file(const char *datafile)
{
    ifstream f;
    vector<vector<int>> matrix;

    f.open(datafile);

    if(f.is_open())
    {
        int val, row = 0, col = 0, rows_total;

        f >> rows_total;
        //initialize matrix vector
        matrix.resize(rows_total);

        //input values from file
        for (int i = 0; i < rows_total; i++) {
            for (int j = 0; j < rows_total; j++) {
                f >> val;
                matrix[i].push_back(val);
            }

        }
    }else
        cout << "\n\tNie ma takiego pliku.\n";

    f.close();

    return matrix;
}
```

- Generowanie danych losowych:
  Użytkownik proszony jest o podanie rozmiaru. Na podstawie wpisanej wartości typu *integer* generowana jest losowa macierz o podanych wymiarach, która jest zwracana jako wektor wektorów typu *integer* w strukturze *Generator* znajdującej się w pliku źródłowym *main.cpp*.

```cpp
static std::vector<std::vector<int>> generate_random_matrix(int rows_total)
{
    //initialize matrix veŚctor
    vector<vector<int>> matrix(rows_total);

    for (int i = 0; i < rows_total; i++) {
        for (int j = 0; j < rows_total; j++) {
            if(i==j)
            {
                matrix[i].push_back(0);
                continue;
            }
            matrix[i].push_back(get_rand_int(10,70));
        }
    }

    return matrix;
}
```

- Metoda pomiaru czasu:
  Do pomiaru czasu zaimplementowana została struktura *Timer*, znajdująca się w pliku źródłowym *main.cpp*.

Została napisana przy pomocy biblioteki *chrono*, która pozwala na dokładne odmierzanie czasu z dokładnością do 1ns. Aby zacząć pomiar należy wywołać Timer.start();, aby zakończyć: Timer.stop(), aby wypisać informacje o długości pomiaru w milisekundach: Timer.info(). Struktura prezentuje się następująco:

```
struct{
    chrono::high_resolution_clock::time_point start_time;
    chrono::high_resolution_clock::time_point stop_time;
    uint64_t duration = -1;

    void start()
    {
        start_time = chrono::high_resolution_clock::now();
    }

    void stop()
    {
        stop_time = chrono::high_resolution_clock::now();
        duration =
        ↪  chrono::duration_cast<chrono::nanoseconds>(stop_time-start_time).count();
    }

    void info()
    {
        cout << "time: " << float(duration)/1000000.0 << "ms" <<endl;
    }
}Timer;
```

- Plan wykonywania pomiarów:
  Wykonywane będą testy na plikach testowych zawartych na stronie prowadzącego. Oba algorytmy zostaną sprawdzone pod kątem czasu oraz rozmiaru danych wejściowych. Sprawdzona zostanie poprawność wykonanego algorytmu, która będzie sprawdzana na podstawie pliku z poprawnymi wynikami. Porównane zostaną czasy wykonywania algorytmów. Celem zwiększenia wiarygodności wykonywanych pomiarów postanowiono dla każdej pary (algorytm, plik testowy) wykonać 50 pomiarów czasu, a wynik uśrednić, za wyjątkiem dużych wartości rozmiarów macierzy dla algorytmu Brute Force - w takich przypadkach pomiar został wykonany raz (ze względu na bardzo długi czas wykonywania). Macierz wejściowa powinna zawierać wartości od 0 do 100. Przykładowa funkcja do uśredniania wynikółw 50 pomiarów:

```
case '6':
            for (int i = 0; i < 50; i++)
            {
                pBranchAndBound = new BranchAndBound(matrix);
                Timer.start();
                pBranchAndBound->find_best_path();
                avg += Timer.stop_test();
            }
            avg/=50.0;
            cout << "usredniony wynik 50 pomiarow to: " << avg/1000000.0 <<"[ms]\n";

            break;

            //for this purpose, there was added function to
            //get one cycle duration in Timer struct.

uint64_t stop_test()
    {
        stop_time = chrono::high_resolution_clock::now();
```

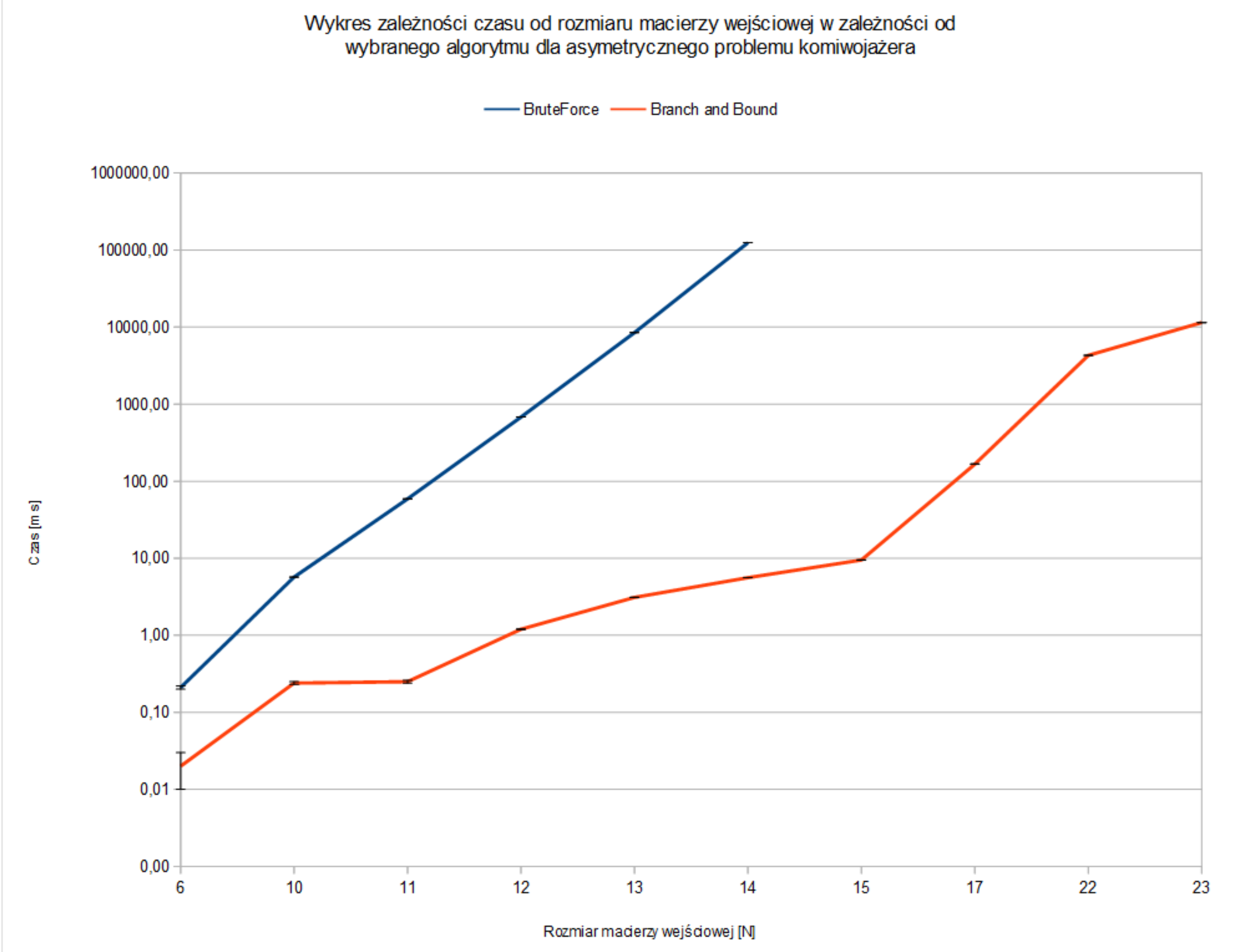
```
            duration =
             ↪  chrono::duration_cast<chrono::nanoseconds>(stop_time-start_time).count();
            return duration;
        }
```

# 5 Wyniki eksperymentów

Wyniki zostały zaokrąglone do dwóch cyfr znaczących. Dla rozmiarów: 6,10,11,12,13,14,15,16 użyte zostały pliki testowe udostępnione przez Prowadzącego. Dla rozmiarów 17,22,23 natomiast zostały wygenerowane losowo wartości z zakresu (10,70).

Tablica 2: Wyniki pomiarów czasów wykonywaniwa implementowanych algorytmów w zależności od rozmiaru macierzy wejściowej

| Rozmiar danych wejściowych | Brute Force [ms] | B&B [ms] |
|---|---|---|
| 6 | 0.21 | 0.02 |
| 10 | 5.70 | 0.24 |
| 11 | 59.00 | 0.25 |
| 12 | 680.00 | 1.20 |
| 13 | 8500.00 | 3.10 |
| 14 | 124413.00 | 5.60 |
| 15 | zbyt długi czas | 9.50 |
| 17 | zbyt długi czas | 167.00 |
| 22 | zbyt długi czas | 4303.00 |
| 23 | zbyt długi czas | 11493.00 |




Rysunek 1: Wykres zależności czasu od rozmiaru macierzy wejściowej w zależności od wybranego algorytmu dla asymetrycznego problemu komiwojażera

# 6 Wnioski dotyczące otrzymanych wyników

Wszystkie wyniki testów są zgodne z arkuszem odpowiedzi, poza parą (B&B, tsp_17.txt), prawdopodobnie ma to związek z postacią macierzy wejściowej - spora ilość zer, która mogła zmylić program. Jednak dla losowo wygenerowanych danych o rozmiarze 17 i każdych innych algorytmu działają bez zarzutów. Zgodnie z Tablicą (2) oraz Wykresem (1) czas wykonywania algorytmu przeglądu zupełnego (Brute Force) jest znacznie dłuższy niż czas wykonywania algorytmu zoptymalizowanego czasowo - algorytmu podziału i ograniczeń (B&B). O ile dla małych wartości rozmiaru wejściowej macierzy kosztów ($N < 7$) czas jest porównywalny, to dla każdej kolejnej wartości czas wykonywania algorytmu przeglądu zupełnego rośnie. Dzieje się tak ze względu na różnice złożoności obliczeniowych obu algorytmów. BruteForce dla większych wartości rozmiaru macierzy wejściowej jest bezużyteczny czasowo. Algorytm podziału i ograniczeń jest natomiast wydajny również dla większych N, co sprawdzono w osobnych testach dla losowo wygenerowanych danych z zakresu (10,70) o rozmiarze $N = 22$ algorytm trwał 4.3[s], a dla $N = 23$ już 11.5[s]. Oznacza to, że w okolicach tych wartości N algorytm przestaje być bardzo wydajny. Jest to jednak nadal nieporównywalny wynik w porównaniu do Brute Force.

# 7 Kod źródłowy

Kod źródłowy został udostępni Prowadzącemu we wskazanym miejscu. Dodatkowo został udostępniony na profil github pod adresem: https://github.com/Sevelantis/pea-p-1.

# 8 Literatura


[1] © Wikipedia Wolna Encyklopedia - Problem Komiwojażera, data dostępu: 14.11.2020, https://pl.wikipedia.org/wiki/Problem_komiwojażera

[2] © Copyright 2018, Antti Salonen - progbook.org, data dostępu: 14.11.2020, https://progbook.org/tsp.html

[3] "On the Computational Complexity of Branch and Bound Search Strategies" by Douglas R. Smith November 1979, Prepared for: National Science Foundation Washington, D. C. 20550. Data dostępu: 14.11.2020, https://apps.dtic.mil/dtic/tr/fulltext/u2/a081608.pdf

[4] 5th Floor, A-118, Sector-136, Noida, Uttar Pradesh - 201305 - GeeksforGeeks, data dostępu: 14.11.2020, https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/