Projektowanie Efektywnych Algorytmów: projekt

28.11.2023

Michał Rzepka

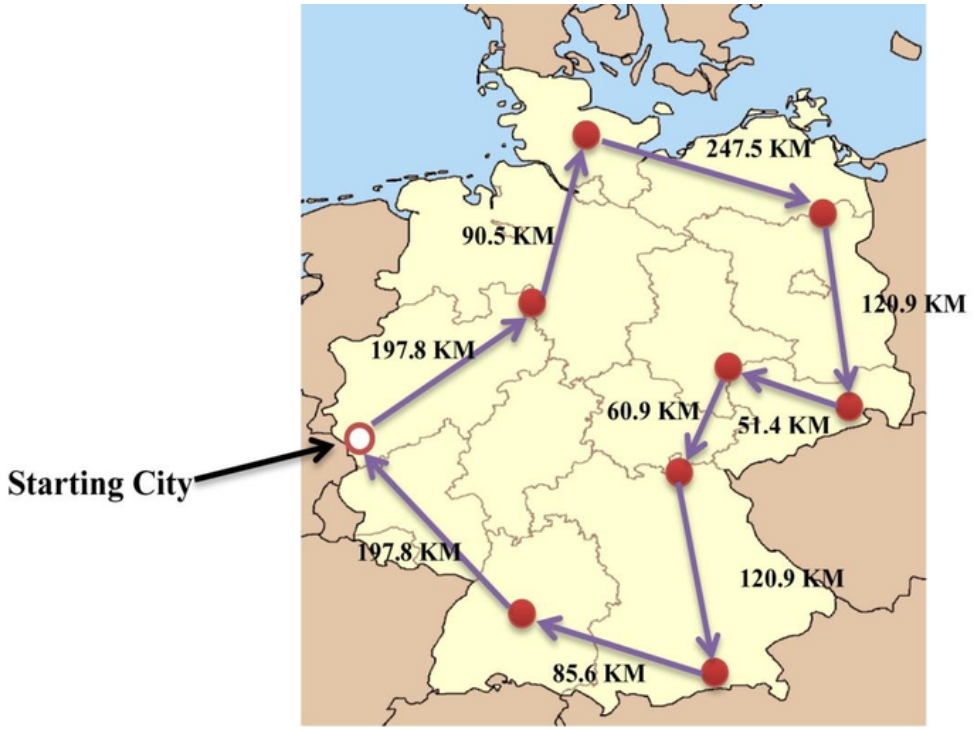
Branch and Bound

Spis treści:

1. Sformułowanie zadania:

Zadanie polega na opracowaniu, implementacji i zbadaniu efektywności algorytmu “Przeglądu zupełnego” rozwiązującego problem Komiwojażera w wersji optymalizacyjnej.

Problem Komiwojażera to zadanie optymalizacyjne polegające na znalezieniu minimalnego (lub maksymalnego) w pełnym grafie ważonym. Cykl Hamiltona to cykl w którym wszystkie wierzchołki grafu są odwiedzone dokładnie raz.



Rysunek 1: Przykładowe rozwiązanie problemu komiwojażera dla 9 miast. Zaznaczone krawędzie należą do minimalnego cyklu Hamiltona, każde miasto jest odwiedzane dokładnie raz

2. Metoda:

Metoda Branch and Bound polega na liczeniu najlepszej odpowiedzi dla konkretnego poddrzewa dla danego węzła/liścia w drzewie. Jeżeli obliczona odpowiedź byłaby gorsza od obecnej najlepszej odpowiedzi, dane poddrzewo jest ignorowane - eliminując tym samym wiele niepotrzebnych obliczeń w stosunku do przeglądu zupełnego. Proces ten w problemie komiwojażera zachodzi poprzez obliczenie wagi danej drogi wykorzystując wcześniej obliczony koszt dostania się do danego wierzchołka grafu od wierzchołka początkowego (odpowiadającego korzeniowi drzewa) oraz obliczoną granicę wagi od obecnego do kolejnego wierzchołka. Jeżeli suma ma większą wagę od obecnie najlepszej odpowiedzi cały podgraf (odpowiadający poddrzewu odpowiedzi) zostaje odrzucony i ignorowany.

Najważniejszymi składowymi metody jest funkcja obliczająca granicę dla węzła oraz strategia odwiedzania wierzchołków w grafie.

Funkcja obliczająca granicę:

- Każdy wierzchołek w drodze jest odwiedzany tylko raz
- Do każdego wierzchołka algorytm droga musi “wejść” i z niego “wyjść” dokładnie raz
- Obliczana jest minimalna waga drogi wejścia do wierzchołka, oraz minimalna waga drogi wyjścia z wierzchołka
- Obliczone minima są dodawane oraz dzielone na 2
- Otrzymana jest dana formuła dla funkcji obliczania granicy: (dla klarowności formuła przedstawiona jest dla pewnego grafu o 4 wierzchołka i liczona dla wierzchołka numer 2)

$$V2bound = \frac{min(V1{->}V2, V3{->}V2, V4{->}V2) + min(V2{->}V1, V2{->}V3, V2{->}V4)}{2}$$

Źródło

3. Algorytm:

Algorytm składa się z kilku elementów:

1. Funkcja inicjalizująca:

```
void TSP(int graf[tabSize][tabSize], int size)
{
    int curr_path[tabSize]; // obecna ścieżka
    int curr_bound = 0; // obecne ograniczenie
    for(int i = 0; i < size+1; i++)
        curr_path[i] = -1;
    for(int i = 0; i < size+1; i++)
        visited[i] = false;
    // Kalkulacja pierwszego ograniczenia
    for (int i=0; i < size; i++)
        curr_bound += (minIncoming(graf, i, size) + minOutgoing(graf, i, size))/2;

    visited[0] = true; // zaczynamy w wierzhołku 0
    curr_path[0] = 0;
    TSPCycle(graf, curr_bound, 0, 1, curr_path, size);
    return;
}
```

2. Funkcje wspomagające - liczące minimum wchodzące do wierzchołka i wychodzące z wierzchołka:

```
int minIncoming(int adj[tabSize][tabSize], int currentVertex, int size)
{
    int min = INT_MAX;
    for(int i = 0; i < size; i++)
    {
        if(adj[i][currentVertex] < min && adj[i][currentVertex] > 0)
            min = adj[i][currentVertex];
    }
    return min;
}
// Funkcja szukająca pierwszego minimum w danym wierzchołku
int minOutgoing(int adj[tabSize][tabSize], int currentVertex, int size)
{
    int min = INT_MAX;
    for (int i = 0; i < size; i++)
        if (adj[currentVertex][i] < min && adj[currentVertex][i] > 0) // Jeżeli droga istnieje
i jest mniejsza od minimum -> ustawiamy minimum na daną drogę
            min = adj[currentVertex][i];
    return min;
}
```

3. Główna pętla algorytmu:

```
void TSPCycle(int adj[tabSize][tabSize], int curr_bound, int curr_weight, int level, int
curr_path[], int size)
{
    if (level == size) // Jeżeli przeszliśmy przez wszystkie wierzchołki w danej iteracji
(liczba cykli jest równa ilości wierzchołków) aktualizujemy ostateczne rozwiązanie i powracamy
    {
        if (adj[curr_path[level-1]][curr_path[0]] != 0) // upewniamy się że istnieje droga od
końcowego do początkowego wierzchołka (czyli jest to cykl hamiltona)
        {
            int curr_res = curr_weight + adj[curr_path[level-1]][curr_path[0]]; // dodajemy
wagę drogi między ostatnim a początkowym wierzchołkiem
            if (curr_res < final_res) // jeżeli otrzymana waga jest mniejsza od wagi
ostatecznego rozwiązania - nadpisujemy ostatecze rozwiązanie
            {
                for (int i=0; i < size; i++)
                    final_path[i] = curr_path[i];
                final_path[size] = curr_path[0];
                final_res = curr_res;
            }
        }
        return;
    }
    for (int i=0; i < size; i++)
    {
        if (adj[curr_path[level-1]][i] > 0 && visited[i] == false) // sprawdzamy każdą
istniejącą drogę do nieodwiedzonego wierzchołka wychodzącą od obecnego wierzchołka
        {
            int temp = curr_bound;
            curr_weight += adj[curr_path[level-1]][i];
            curr_bound -= (minIncoming(adj, i, size) + minOutgoing(adj, i, size))/2;
            if (curr_bound + curr_weight < final_res)
            {
                curr_path[level] = i;
                visited[i] = true;
                TSPCycle(adj, curr_bound, curr_weight, level+1, curr_path, size);
            }
            // W przypadku gdy nie znaleziono mniejszego ograniczenia lub wracamy resetujemy
wszystkie potencjalnie zmienione informacje
            curr_weight -= adj[curr_path[level-1]][i]; // reset wagi
            curr_bound = temp;                         // reset ograniczenia
            for (int i = 0; i < size; i++) // reset tablicy wizyt do stanu obecnej ścieżki
                visited[i] = false;
            for(int i = 0; i < level; i++)
                visited[curr_path[i]] = true;
        }
    }
}
```

Przykład działania algorytmu dla danego grafu o 4 wierzchołkach:

```
-1  81  50  18
81  -1  76  21
50  76  -1  24
18  21  24  -1
```

Odpowiedź dla grafu: 165

Informacja autora: Nie jestem w stanie wytłumaczyć dlaczego mój algorytm działa poprawnie. Według moich wielu prób debugowania, modyfikacji i przechodzenia krok po kroku razem z algorytmem - jego działanie jest według mnie błędne. Jednak przechodzi on każdy test na instancjach o znanym przeze mnie wyniku. W dodatku im bardziej niepoprawnie wygląda dla mnie struktura jego wykonania tym szybciej oblicza on operacje na dużych instancjach. Wersja algorytmu która była najszybsza na instancjach powyżej 16 (jest 27 razy szybsza od algorytmu będącego według mnie bardziej poprawnym dla instancji wielkości 17) wykonała także przegląd zupełny dla powyższego grafu o wielkości instancji 4, mimo mojego przekonania że jest to działanie nieprawidłowe. W tym wypadku pozwoliłem sobie podarować przejście krok po kroku z algorytmem gdyż wykonanie przeglądu zupełnego mija się z celem prezentacji algorytmu branch and bound. Moimi jedynymi hipotezami są złe zrozumienie działanie poprawnego algorytmu z powodu korzystania ze złych źródeł, lub zła konstrukcja instancji testowych. Obliczenia czasowe oparte na działaniu najszybszej wersji algorytmu która jest także opisana powyżej.

Źródła/inspiracje:

Szkielet i format algorytmu

Działanie funkcji obliczających granicę oraz dokładny obraz działania algorytmu

4. Dane testowe:

Do pomiaru dokładności oraz czasu wykonania algorytmu wykorzystano dany zestaw instancji

Nazwa instancji	Waga optymalnej drogi
Dane4.txt (tylko do dokładności)	165
Dane6.txt (tylko do dokładności)	150
Dane8.txt (tylko do dokładności)	136
Dane11.txt (tylko do dokładności)	149
Dane12.txt	-
Dane13.txt	-
Dane14.txt	-
Dane15.txt	-
Dane16.txt (do dokładności)	82
Dane17.txt (do dokładności)	2085
Dane18.txt	-
Dane19.txt (do dokładności)	2413
Dane20.txt	-
Dane21.txt	-
Dane22.txt	1216

Pliki z danymi testowymi zostały umieszczone na repozytorium: https://github.com/Michal13241/PEA-pliki-testowe

5. Procedura badawcza

a. Pliki konfiguracyjne i wejściowe

Program po zainicjalizowaniu ładuje dane z pliku *config.cfg*. Format pliku to:

- Linia z jedną liczbą n – liczbą testów dla danego zestawu danych
- Kolejna linia ze ścieżką do pliku zawierającego dane dla danej instancji

Format pliku wejściowego zawierającego dane dla danej instancji:

- Linia z jedną liczbą n - wielkością instancji
- N linii po n liczb

Format pliku wyjściowego:

- Linia z jedną liczbą n - liczba wyników czasowych zmierzonych dla podanej instancji
- N linii z wynikami dla testów
- Linia ze średnią powyższych wyników

Przykład wyglądu zapisanego pliku dla 6 zestawów danych:

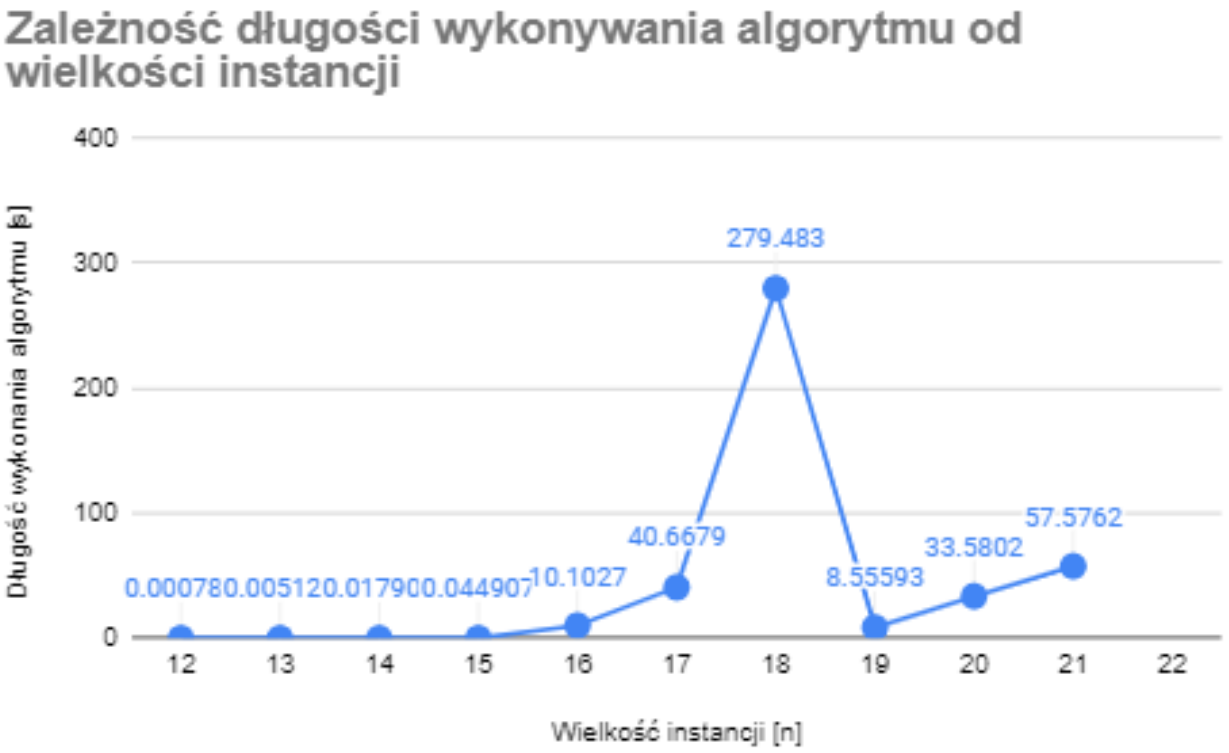
b. Pomiar czasu

Program testowany był na podsystemie linux zawartym w windowsie “windows subsystem for linux” z procesorem Intel Core i7-7700HQ. Mierzony został czas od wywołania funkcji algorytmu dla jednego testu dla jednego zestawu danych do końca wykonania danej funkcji zawierającej główny algorytm. Fragmenty kodu odpowiadające za pomiary czasu:

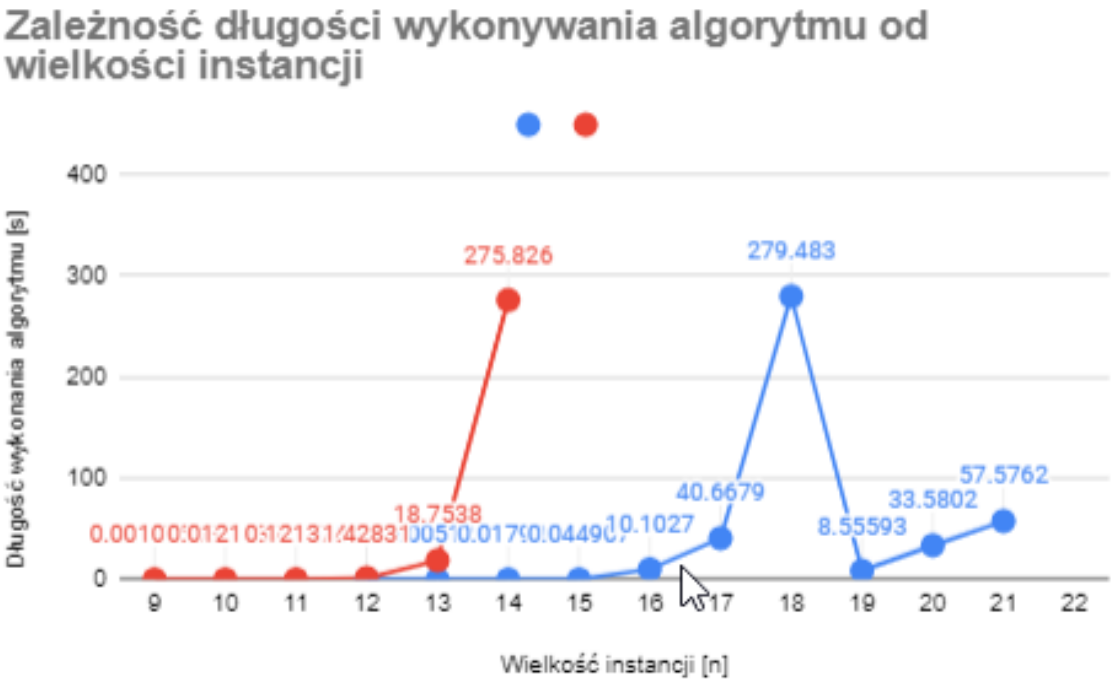
```
    clock_t time, timesum = 0;
    filename = "wyniki/czasy.txt";
    ofstream of_plik;

    of_plik.open(filename, ios::in | std::ofstream::trunc);
    of_plik << numberOfTests << "\n";
    for(int i = 0; i < numberOfTests; i++)
    {
        time = test(graf, size);
        timesum += time;
        of_plik << ((float)time)/CLOCKS_PER_SEC << "\n";
    }
    of_plik << ((float)timesum)/(numberOfTests*CLOCKS_PER_SEC);
    of_plik.close();
```

```
clock_t test(int graf[tabSize][tabSize], int size)
{
    clock_t time;
    time = clock();
    final_res = INT_MAX;
    TSP(graf, size);
    return clock() - time;
}
```

6. Wyniki



Rysunek 2: Wykres zależności długości wykonywania algorytmu od wielkości instancji



Rysunek 3: Wykres porównujący długości wykonywania algorytmu dla metody przeglądu zupełnego (czerwony) oraz Branch and bound (niebieski)

7. Analiza wyników

Krzywa wzrostu czasu ma ogólny charakter wykładniczy. Istnieją jednak poważne skoki między pewnymi instancjami. Zgadza się to z naturą algorytmu branch and bound będącego bardzo zależnego od wyglądu instancji na której wykonywany jest algorytm. Na powyższym można zauważyć znaczące anomalie między:

- 15 a 16 instancją (bardzo duże zwiększenie czasu instancji)
- 16 a 17 instancją (skok jedynie czterokrotny mimo ~200 razy dłuższego czasu między instancją 15 a 16)
- 18 a 19 instancją (zmniejszenie się czasu wykonywania algorytmu mimo zwiększenia wielkości instancji)
- 21 a 22 instancją (mimo tego, że algorytm dla 21 instancji wyniósł niewiele więcej niż dla 17, algorytm dla 22 instancji nie wykonała się nawet raz w przeciągu ponad 30 minut)

Przypuszczenia wobec działania branch and bound mogą być potwierdzone analizując poszczególne instancje:

- Instancja 16 i 18 są do siebie bardzo podobne, zostały też intencjonalnie tworzone w celu utrudnienia obliczeń dla algorytmów heurystycznych, do których w pewnym stopniu można zaliczyć BnB
- Instancja 17 jest standardową instancją
- Instancje 19, 20 i 21 są do siebie bardzo podobne - różniąc się jedynie jednym dodanym wierzchołkiem (21 zawiera w sobie 20, a 20 zawiera w sobie 19)
- Instancja 22 jest standardową instancją

Wszystkie informacje zgadzają się ze specyfikacją metody Branch and Bound. W najgorszym wypadku funkcja ma charakter wykładniczy i w teorii może wykonać przegląd zupełny - jednak statystycznie złożoność tej funkcji jest niższa i nieokreślona.

Porównując metodę BnB do przeglądu zupełnego można zauważyć podobieństwo wyglądu wykresu po zignorowaniu anomalii metody BnB. Jednak niezaprzeczalnie przegląd zupełny jest mniej efektywnym algorytmem dla zwiększających się instancji. Standardowa złożoność metody przeglądu zupełnego jest jedynie najgorszym możliwym przypadkiem dla metody Branch and Bound.