Projektowanie efektywnych algorytmów

Projekt

15.11.2020

248821 Michał Rajkowski

(3)Branch&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 (ang. Travelling salesman problem) jest to zagadnienie optymalizacyjne polegające na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym.

Zagadnienie to można podzielić na dwa podtypy ze względu na wagi ścieżek łączących wierzchołki grafu. W symetrycznym problemie komiwojażera (STSP) dla dowolnych dwóch wierzchołków A B krawędź łącząca wierzchołek A z B ma taką samą wagę jak ta łącząca wierzchołek B z A. Natomiast w asymetrycznym problemie komiwojażera (ATSP) wagi te mogą się różnić.

Główną trudnością problemu jest duża liczba danych do analizy. W przypadku problemu symetrycznego dla n miast liczba kombinacji wynosi $\frac{(n-1)!}{2}$.

Dla problemy asymetrycznego jest to aż n!.

2. Metoda

Branch & Bound (BnB) jest metodą rozwiązywania problemów NP-trudnych. W oparciu o nią algorytmy rozpatruje wszystkie możliwe rozwiązania problemu, dzieląc pośrednie rozwiązania na podproblemy które tworzą strukturę drzewa. Nierozwiązane jeszcze podproblemy są rozwiązywane w sposób rekurencyjny powiększając strukturę drzewa (tzw. branching) jednakże algorytm także nie rozpatruje sub-optymalnych gałęzi drzewa, gdyż wie że na ich końcu nie będzie znajdować się optymalne rozwiązanie (tzw. bounding). Gdy algorytm rozpatrzy już całe drzewo, najlepsze rozwiązanie znalezione przez BnB będzie rozwiązaniem optymalnym.

Na wydajność algorytmu mają wpływ 3 główne czynniki:

- strategia przeszukiwania – kolejność w jakiej sub-problemy w drzewie są eksplorowane
- strategia podziału – jak przestrzeń rozwiązań jest dzielona aby wytworzyć nowe sub-problemy drzewa
- zasady okrajania – zasady które zapobiegają zagłębiania się w suboptymalne rejony drzewa

Poniżej zarysowana jest typowa struktóra działania algorytmów BnB:

Algorithm 1: Branch-and-Bound(X, f)

1 Set $L = \{X\}$ and initialize $\hat{x}$
2 **while** $L \neq \emptyset$**:**
3 Select a subproblem S from L to explore
4 **if** a solution $\hat{x}' \in \{x \in S \mid f(x) < f(\hat{x})\}$ can be found**:** Set $\hat{x} = \hat{x}'$
5 **if** S cannot be pruned**:**
6 Partition S into $S_1, S_2, \ldots, S_r$
7 Insert $S_1, S_2, \ldots, S_r$ into L
8 Remove S from L
9 Return $\hat{x}$

Źródło: https://www.sciencedirect.com/science/article/pii/S1572528616000062

Na złożoność obliczeniową algorytmu wpływają wyżej wymienione czynniki. Jednakże bez względu na to jak zostaną dobrane w pesymistycznym przypadku algorytm degeneruje się do przeszukiwania siłowego O(n!). W praktyce czas znalezienia rozwiązania zależy w dużej części od grafu danych wejściowych, ale nie jedynie ilości jego wierzchołków a dysproporcji między krawędziami. Jeżeli posiadają one podobne długości to nasze obcinanie nieoptymalnych rozwiązań będzie zachodziło tuż przy liściach drzewa.

3. Algorytm

3.1 Komentarz do zastosowanych rozwiązań

Strategią przeszukiwania przestrzeni rozwiązań w opracowanym przeze mnie kodzie jest przeszukiwanie w głąb (Depth search).

Przy rozpoczęciu swojego działania algorytm nie wylicza w sposób heurystyczny wartości według której odrzucane będą stany suboptymalne. Rozwiązanie to na początku działania algorytmu wynosi nieskończoność i jest nadpisywane przypadku każdorazowego dotarcia do liści drzewa i znalezienia lepszego rozwiązania problemu TSP na danej instancji. Rozwiązanie to nazwijmy B.

Algorytm ucina suboptymalne rozwiązania drzewa gdy zachodzi nierówność:

$$B < LB(Xj)$$

Gdzie LB oznacza Lower Bound i jest równe:

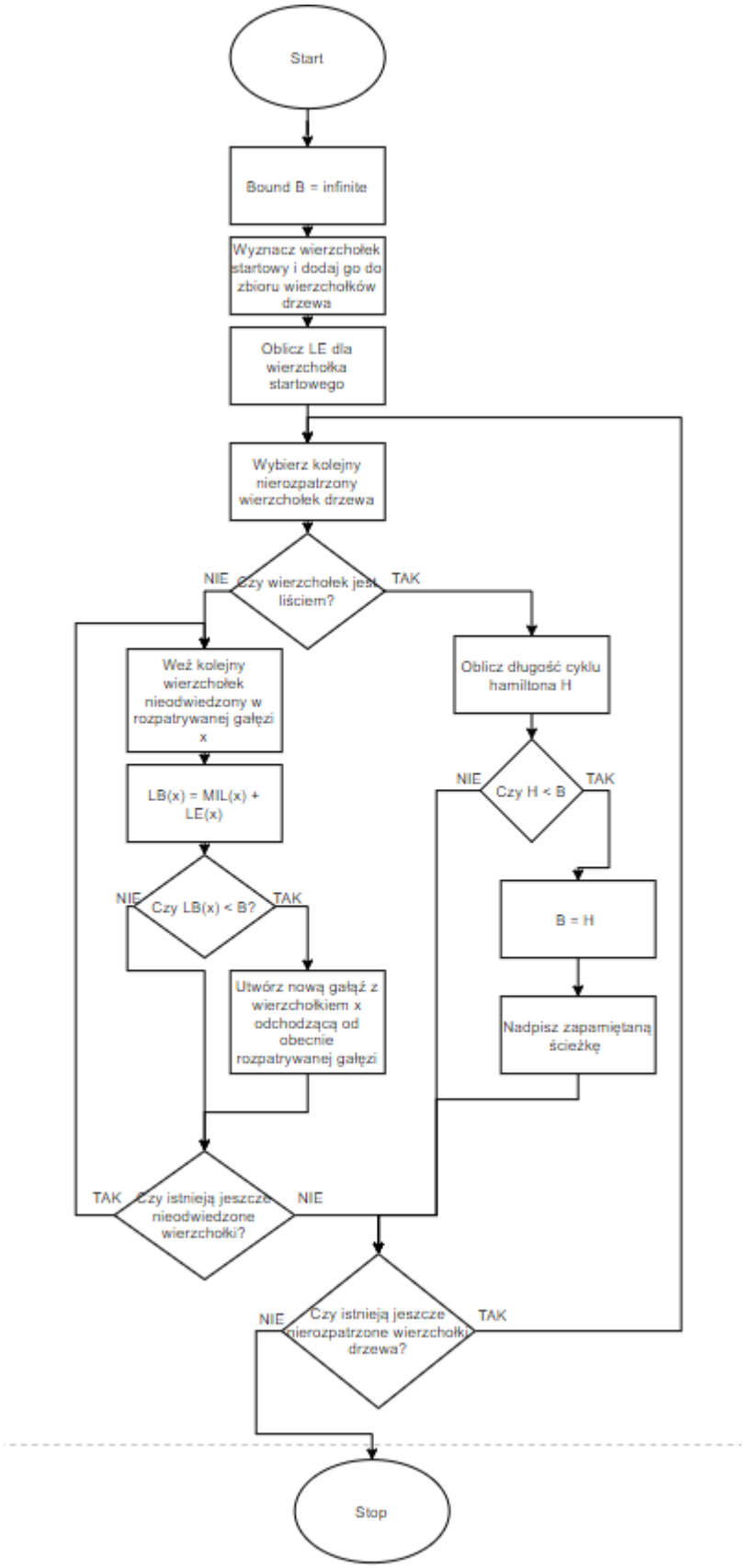
$$LB(Xj) = MIL(Xj) + LE(Xj).$$

MIL oznacza sumaryczny przebyty dla danego stanu dystans. Natomiast LE (Lower Estimate) jest najniższą estymowaną wartością jaką może przyjąć pozostała do przebycia droga.

Obrany przeze mnie sposób estymacji LE polega na obliczaniu dla każdego pozostałego wierzchołka sumy najtańszych krawędzi wchodzących i wychodzących i podzielenia uzyskanego wyniku przez 2. Uzyskujemy w ten sposób estymowany najtańszy koszt przejścia po pozostałych wierzchołkach.

Dobranie dobrego sposobu estymacji pozostałej ścieżki jest trudnym zadaniem. W teorii im wyższe daje ona wyniki, tym więcej zbędnych gałęzi nasz algorytm będzie ucinał i wpłynie to znacząco na szybkość jego działania. Jednakże bardziej skomplikowane sposoby obliczania LE trwają dłużej w wyniku czego dla każdego wierzchołka drzewa program będzie potrzebował więcej czasu na wykonanie obliczeń. Widzimy zatem iż dobieranie sposobu wyliczania LE jest tak naprawdę próbą znalezienia balansu pomiędzy ilością ucinanych wierzchołków a czasem ich przetwarzania w celu zminimalizowania czasu działania algorytmu.

3.2 Schemat blokowy



3.3 Kod

```
int findEnter(int node){
    int min = INT_MAX;
    for(int i = 0; i < cnt; i++){
        if(i == node)
            continue;
        if(tab[node][i] < min)
            min = tab[node][i];
    }
    return min;
}
int findLeave(int node){
    int min = INT_MAX;
    for(int i = 0; i < cnt; i++){
        if(i == node)
            continue;
        if(tab[i][node] < min)
            min = tab[node][i];
    }
    return min;
}

//Algorithm global space end

void Recursion(int LE, int MIL, int depth){
    if(depth == cnt){

        int result = MIL + tab[path[depth-1]][0];

        if(result < opt){
            opt = result;
            for(int i = 1; i < cnt; i++){
                path_final[i] = path[i];
            }
        }

        return;
    }

    for(int i = 0; i < cnt; i++){
        if(i == path[depth - 1] || visited[i] == true)
        continue;

        int new_MIL = MIL;
        int new_LE = LE;
        new_MIL += tab[path[depth - 1]][i];
        new_LE -= (findEnter(i) + findLeave(path[depth - 1])) /2;

        if(new_LE + new_MIL < opt){
            path[depth] = i;
            visited[i] = true;

            Recursion(new_LE, new_MIL, depth + 1);
        }
        visited[i] = false;


    }

}

int TSP_BnB(){

    path = new int[cnt];

    opt = INT_MAX;
    int MIL = 0;
    int depth = 0;

    path_final = new int[cnt + 1];

    visited = new bool[cnt];

    int LE = 0;

    for(int i = 0; i < cnt; i++){
        visited[i] = false;
    }
    visited[0] = true;

    for(int i = 0; i < cnt; i++){
        LE += findEnter(i);
        LE += findLeave(i);
    }
    LE = LE / 2;

    path[0] = 0;
    path[cnt] = 0;

    Recursion(LE, MIL, depth + 1);

}
```

4. Dane testowe

Do sprawdzenia poprawności działania algorytmu oraz do badań wybrano następujący zestaw instancji:

- tsp_6_1.txt
- tsp_10.txt
- tsp_12.txt
- tsp_13.txt
- tsp_14.txt
- tsp_17.txt
- gr17.txt
- gr21.txt
- gr24.txt
- br17.txt

tsp_6_1.txt - tsp_17.txt to pliki testowe pochodzący ze strony doktora Mierzwy:

http://jaroslaw.mierzwa.staff.iiar.pwr.wroc.pl/pea-stud/tsp/

Pliki gr17.txt – gr24.txt oraz br17.txt zostały utworzone na bazie stron:

http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html

http://elib.zib.de/pub/mp-testdata/tsp/tsplib/atsp/index.html

i zostały okrojone o niepotrzebne nagłówki pliku aby łatwiej było obsługiwać je w programie, stąd rozszerzenie txt.

5. Procedura badawcza

Należało zbadać zależność czasu rozwiązania problemu od wielkości instancji. W przypadku algorytmu realizującego branch and bound nie występowały parametry programu, które mogły mieć wpływ na czas i jakość uzyskanego wyniku. W związku z tym procedura badawcza polegała na uruchomieniu programu sterowanego plikiem inicjującym .INI (format pliku: nazwa_instancji liczba_wykonań rozwiązanie_optymalne [ścieżka optymalna]; nazwa_pliku_wyjściowego).

tsp_10.txt 20 212 0 3 4 2 8 7 6 9 1 5 0

tsp_12.txt 20 264 0 1 8 4 6 2 11 9 7 5 3 10 0

tsp_13.txt 20 269 0 10 3 5 7 9 11 2 6 4 8 1 12 0

tsp_14.txt 20 282 0 10 3 5 7 9 13 11 2 6 4 8 1 12 0

tsp_17.txt 1 39 16 8 7 4 3 15 14 6 5 12 10 9 1 13 2 11 0 16

br17.txt 1 39 16 8 7 4 3 15 14 6 5 12 10 9 1 13 2 11 0 16

gr17.txt 5 2085 16 13 14 2 10 9 1 4 8 11 15 0 3 12 6 7 5 16

gr21.txt 5 2707 20 14 13 12 17 9 16 18 19 10 3 11 0 6 7 5 15 4 8 2 1 20

gr24.txt 5 1272 23 7 20 4 9 16 21 17 18 14 1 19 13 12 8 22 3 11 0 15 10 2 6 5 23

final_test.csv

Każda instancji rozwiązywana była zgodnie z liczbą jej wykonań, np. tsp_10.txt wykonana została 20 razy. Do pliku wyjściowego final_test.csv zapisywany był czas wykonania, otrzymane rozwiązanie (koszt ścieżki) oraz ścieżka (numery kolejnych węzłów). Plik wyjściowy zapisywany był w formacie csv. Poniżej przedstawiono fragment zawartości pliku wyjściowego:

tsp_12.txt 20 264 0 1 8 4 6 2 11 9 7 5 3 10 0

5

5

6

5

(…)

tsp_13.txt 20 269 0 10 3 5 7 9 11 2 6 4 8 1 12 0

28

30

29

29

(…)

5.1 Metoda pomiaru czasu

Pomiary czasu uzyskane zostały przy pomocy funkcji clock pochodzącej z biblioteki time.h.

Funkcja clock zwraca czas zużyty przez procesor w „clock ticks”. Następnie otrzymane tiki zostają przekonwertowane na milisekundy przy pomocy makra CLOCKS_PER_SEC. Za ostateczne obliczanie czasu odpowiedzialna jest następująca linia kodu:

```
cpu_time = ((double) (end - start)) / (CLOCKS_PER_SEC/1000);
```

5.2 Sprzęt

Procesor : Intel Xeon x5450, 4 rdzenie, 3.00 GHz

RAM: 8.00 GB

System: 64-bitowy system operacyjny, procesor x64

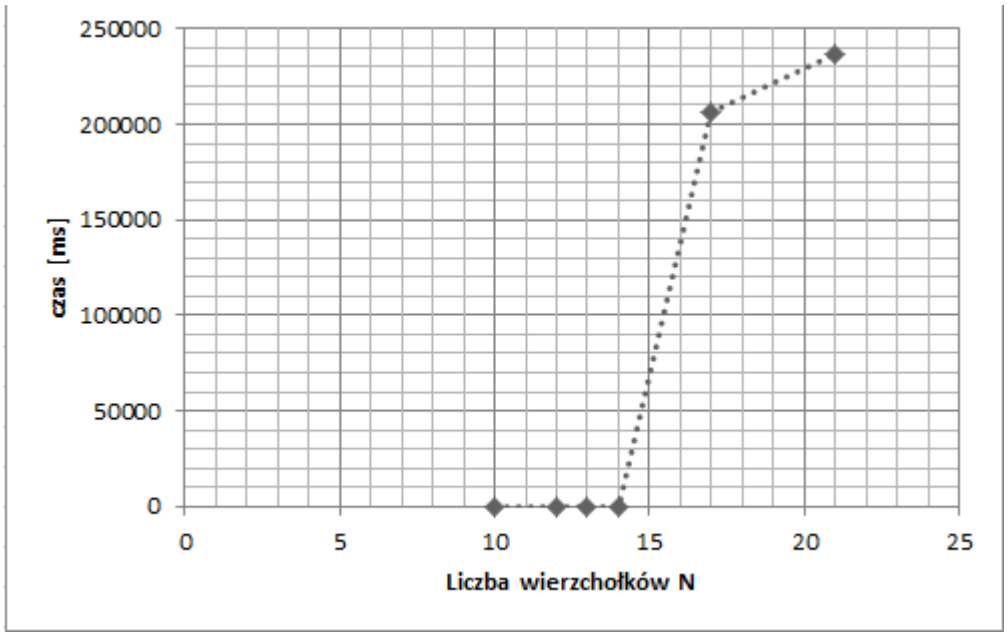
6. Wyniki

6.1 Pomiar czasu

Wyniki zwrócone w wyniku działania programu są przechowane w pliku final_test.csv.(jest to czysta wersja wyników zwróconych przez program). Natomiast ich wersja przedstawiona w przystępniejszej formie, wraz z wykresami została zawarta w pliku wyniki.xlsx.

Dla testów tsp_17.txt, br17.txt oraz gr24.txt czas ich wykonania wynosił ponad 30 minut. Z tego powodu zostały one odrzucone z zestawienia końcowego.

Wyniki obliczeń zostały przedstawione na wykresie (rys 1).



(rys 1.)

7. Wnioski

Obserwując utworzony przez wyniki obliczeń wykres widzimy, iż zachowuje się on w sposób nieregularny. Możemy na tej podstawie domyślać się, że czas rozwiązania zadania zależy nie tylko od liczby wierzchołków grafu TSP.

Wiemy także, iż dla różnych instancji o tej samej liczbie wierzchołków algorytm zwraca drastycznie inne wyniki, np. dla tsp_17 oraz br17 jest to ponad 30 minut, natomiast dla gr17 jedynie kilka minut.

Na tej podstawie możemy domyślać się, iż czas działania algorytmu zależy nie tylko od ilości wierzchołków ale także ułożenia danych wejściowych, to jest wag krawędzi między poszczególnymi wierzchołkami. Pokrywa się to z założeniami teoretycznymi. W przypadku testu tsp_17 oraz br_17 dane wejściowe były tak ułożone że otrzymaliśmy przypadek pesymistyczny i algorytm zdegenerował się do brute force'a.

Z tego powodu ciężko jest określić jaka jest ogólna złożoność obliczeniowa czasowa algorytmu. Wpływa na nią bardzo wiele czynników i nie jest to trywialne zadanie. Wiemy jednak jak działa odcinanie sub-optymalnych gałęzi i wiemy iż następuje to w momencie gdy szacowana minimalna wartość cyklu przekracza już znalezioną. Widzimy zatem iż dala grafów w których przez dłuższy czas nie jesteśmy pewni czy można jeszcze dla danej gałęzi znaleźć optymalny cykl Hamiltona algorytm zbliża się w działaniu do Brute Force.

Jakie są to grafy? Nie posiadam formalnego matematycznego dowodu ale na podstawie przemyśleń oraz testów wydaje się, iż BnB działa tym lepiej im bardziej chaotyczna jest budowa grafu. Gdy zajrzymy do struktury grafów dla których algorytm wykonywał obliczenia najdłużej występują tam małę dysproporcje między wartościami krawędzi oraz krawędzie przyjmują podobne wartości. Dla wygenerowanych przeze mnie testów w których grafy miały duże dysproporcje między wartościami krawędzi i dużą losowość algorytm znajdował rozwązania w bardzo krótkim czasie. Przemawiałoby to za potwierdzeniem tej hipotezy.

Największą korzyścią płynącą z BnB jest jego małe zużycie pamięci. Mimo iż nie dorównuje on w szybkości obliczeń algorytmowi dynamicznemu, to może się okazać lepszym rozwiązaniem dla większych testów (N > 25) gdyż jest to bariera pamięci nie do przejścia dla bardzo kosztownego w pamięć algorytmu Helda-Karpa, a być może dane będą ułożone tak, iż BnB policzy je nie w czasie Brute Force.