Jan Potocki
nr indeksu 226104

SPRAWOZDANIE

Projektowanie efektywnych algorytmów
Projekt nr 1:
Implementacja i analiza efektywności algorytmu podziału i ograniczeń lub programowania dynamicznego dla wybranego problemu optymalizacji
pon 11:15
13.11.2017

1. Informacje teoretyczne

Problem klasy NP jest problemem, dla którego rozwiązanie można zweryfikować w czasie wielomianowym – natomiast samo znalezienie rozwiązania problemu prawie na pewno wymaga czasu ponadwielomianowego – nie znaleziono dotąd algorytmu o wielomianowej złożoności czasowej. Dla rozwiązania problemów klasy NP można zastosować przegląd zupełny – otrzymamy wtedy dokładne rozwiązanie, jednak ze względu na bardzo dużą złożoność czasową, nadaje się on jedynie dla bardzo małych rozmiarów problemu. Dla nieco większych instancji problemów również możemy otrzymać dokładny wynik stosując algorytm oparty na programowaniu dynamicznym lub metodzie podziału i ograniczeń (branch and bound).
Do implementacji w ramach projektu wybrałem metodę podziału i ograniczeń dla problemu komiwojażera.

Asymetryczny problem komiwojażera:
Parametrami zadania są: skończony zbiór miast $C = \{c^1, c^2, \ldots, c^n\}$ oraz odległości d_i z miasta c_i do miasta c_j (nie ma wymogu $d_{ij} = d_{ji}$). Należy określić kolejność odwiedzania wszystkich miast ich permutację) $<c_{i[1]}, c_{i[2]}, \ldots, ci_{[n]}>$, aby sumaryczna trasa była jak najkrótsza przy założeniu, że każde miasto zostało odwiedzone dokładnie jeden raz. Wszystkie parametry zadania są liczbami naturalnymi.
Złożoność obliczeniowa:

- złożoność czasowa przeglądu zupełnego (brute force) wynosi O(n!), gdzie n – ilość miast[1]
- złożoność czasowa algorytmu wykorzystującego metodę podziału i ograniczeń jest bardzo mocno zależna od danych; w pesymistycznym przypadku wynosi O(n!), gdzie n – ilość miast, ale dla większości instancji problemu będzie w stanie znaleźć rozwiązanie szybciej[2]

2. Opis algorytmu

W moim programie zaimplementowałem algorytm oparty na metodzie podziału i ograniczeń opisany w pliku[3], w wersji stosującej strategię przeszukiwania typu najpierw najlepszy. W algorytmie wykorzystana jest kolejka priorytetowa, drzewo przestrzeni stanów reprezentowane jest niejawnie. Ogólna zasada działania jest następująca: zaczynając od korzenia (zawierającego tylko wierzchołek startowy), dla każdego z potomków aktualnie rozpatrywanego węzła w drzewie (częściowego rozwiązania, zawierającego wszystkie dotychczas umieszczone na trasie wierzchołki) policz ograniczenie dolne i jeżeli jest ono lepsze od aktualnego ograniczenia górnego, rozpatruj kolejno potomków w następnych krokach, od najlepszej do najgorszej wartości funkcji ograniczenia dolnego.

Pseudokod opisywanego algorytmu wygląda następująco:

1 https://en.wikipedia.org/wiki/Travelling_salesman_problem (dostęp 12.11.2017)
2 http://www.geeksforgeeks.org/branch-bound-set-5-traveling-salesman-problem/ (dostęp 12.11.2017)
3 https://www.ii.uni.wroc.pl/~prz/2011lato/ah/opracowania/met_podz_ogr.opr.pdf (dostęp 13.11.2017)

```
initialize(Q)
v ← korzeń drzewa
best ← ∞
best_route ← NULL
insert(Q,v)
while !empty(Q) do
        v ← remove(Q)
        if bound(v) jest lepszy od best then
                for all dziecko u węzła v do
                        if u jest liściem then
                                if length(u) jest lepsza od best then
                                        insert(u, 1)
                                        best ← length(u)
                                        best_route ← u
                                end if
                        else
                                if bound(u) jest lepszy od best then
                                        insert(Q,u)
                                end if
                        end if
                end for
        else
                break
        end if
end while
```

Górnym ograniczeniem jest najkrótsza znaleziona dotąd trasa (wyznaczona w liściu drzewa, początkowo nieskończoność). Dolnym ograniczeniem jest suma długości dotychczas przebytej drogi i dolnej granicy ścieżek, których prefiksem jest ścieżka w obecnym węźle. Można wyznaczyć ją w następujący sposób: dla każdego z niewykorzystanych jeszcze wierzchołków obliczamy minimalną długość wychodzących z niego krawędzi, które jeszcze mogą być użyte. Przy czym:

- bierzemy pod uwagę wszystkie krawędzie nie prowadzące do wierzchołków już umieszczonych w częściowym rozwiązaniu
- jeżeli to nie jest ostatni wierzchołek w częściowym rozwiązaniu, bierzemy pod uwagę krawędzie prowadzące do wierzchołka początkowego (mimo, że on zawsze jest w rozwiązaniu)

Przykład:

$$\begin{bmatrix} 0 & 14 & 4 & 10 & 20 \\ 14 & 0 & 7 & 8 & 7 \\ 4 & 5 & 0 & 7 & 16 \\ 11 & 7 & 9 & 0 & 2 \\ 18 & 7 & 17 & 4 & 0 \end{bmatrix}$$

Ilustracja 1: Macierz sąsiedztwa

v = {1}, best = ∞, Q = {0|(1)}

1. Q == {0|(1)} → v = {1}
   v == {1}
   bound(v) == 0 < ∞ → u = {1-2}, nie jest liściem → bound(u) = 14+7+4+2+4=31 < ∞

→ insert(Q,u)
u = {1-3}, nie jest liściem → bound(u) = 4+7+5+2+4=22 < ∞
→ insert(Q,u)
u = {1-4}, nie jest liściem → bound(u) = 10+7+4+2+7=30 < ∞
→ insert(Q,u)
u = {1-5}, nie jest liściem → bound(u) = 20+7+4+7+4=42 < ∞
→ insert(Q,u)

2. Q = {22|(1-3), 30|(1-4), 31|(1-2), 42|(1-5)} → v = {1-3}
v == {1-3}
bound(v) == 22 < ∞ → u = {1-3-2}, nie jest liściem → bound(u) = 4+5+7+2+4=22 < ∞
→ insert(Q,u)
→ u = {1-3-4}, nie jest liściem → bound(u) = 4+7+7+2+7=27 < ∞
→ insert(Q,u)
→ u = {1-3-5}, nie jest liściem → bound(u) = 4+16+8+7+4=39 < ∞
→ insert(Q,u)

3. Q = {22|(1-3-2), 27|(1-3-4), 30|(1-4), 31|(1-2), 39|(1-3-5), 42|(1-5)} → v = {1-3-2}
v == {1-3-2}
bound(v) == 22 < ∞ → u = {1-3-2-4}, nie jest liściem → bound(u) = 4+5+8+2+18=37 < ∞
→ insert(Q,u)
→ u = {1-3-2-5}, nie jest liściem → bound(u) = 4+5+7+11+4=31 < ∞
→ insert(Q,u)

4. Q = {27|(1-3-4), 30|(1-4), 31|(1-2), 31|(1-3-2-5), 37|(1-3-2-4), 39|(1-3-5), 42|(1-5)}
→ v = {1-3-4}
v == {1-3-4}
bound(v) == 27 < ∞ → u = {1-3-4-2}, nie jest liściem → bound(u) = 4+7+7+7+18=43 < ∞
→ insert(Q,u)
→ u = {1-3-4-5}, nie jest liściem → bound(u) = 4+7+2+14+7=34 < ∞
→ insert(Q,u)

5. Q = {30|(1-4), 31|(1-2), 31|(1-3-2-5), 34|(1-3-4-5), 37|(1-3-2-4), 39|(1-3-5), 42|(1-5),
43|(1-3-4-2)} → v = {1-4}
v == {1-4}
bound(v) == 30 < ∞ → u = {1-4-2}, nie jest liściem → bound(u) = 10+7+7+4+17=
=45 < ∞ → insert(Q,u)
→ u = {1-4-3}, nie jest liściem → bound(u) = 10+9+7+5+7=
=38 < ∞ → insert(Q,u)
→ u = {1-4-5}, nie jest liściem → bound(u) = 10+2+7+4+7=
=30 < ∞ → insert(Q,u)

6. Q = {30|(1-4-5), 31|(1-2), 31|(1-3-2-5), 34|(1-3-4-5), 37|(1-3-2-4), 38|(1-4-3), 39|(1-3-5),
42|(1-5), 43|(1-3-4-2), 45|(1-4-2)} → v = {1-4-5}
v == {1-4-5}
bound(v) == 30 < ∞ → u = {1-4-5-2}, nie jest liściem → bound(u) = 10+2+7+7+4=
=30 < ∞ → insert(Q,u)
bound(v) == 30 < ∞ → u = {1-4-5-3}, nie jest liściem → bound(u) = 10+2+17+14+5=
=48 < ∞ → insert(Q,u)

7. Q = {30|(1-4-5-2), 31|(1-2), 31|(1-3-2-5), 34|(1-3-4-5), 37|(1-3-2-4), 38|(1-4-3), 39|(1-3-5),
42|(1-5), 43|(1-3-4-2), 45|(1-4-2), 48|(1-4-5-3)} → v = {1-4-5-2}
v == {1-4-5}
bound(v) == 30 < ∞ → u = {1-4-5-2-3}, jest liściem → length(u) = 10+2+7+7+4=
=30 < ∞ → best = 30
best_route=u

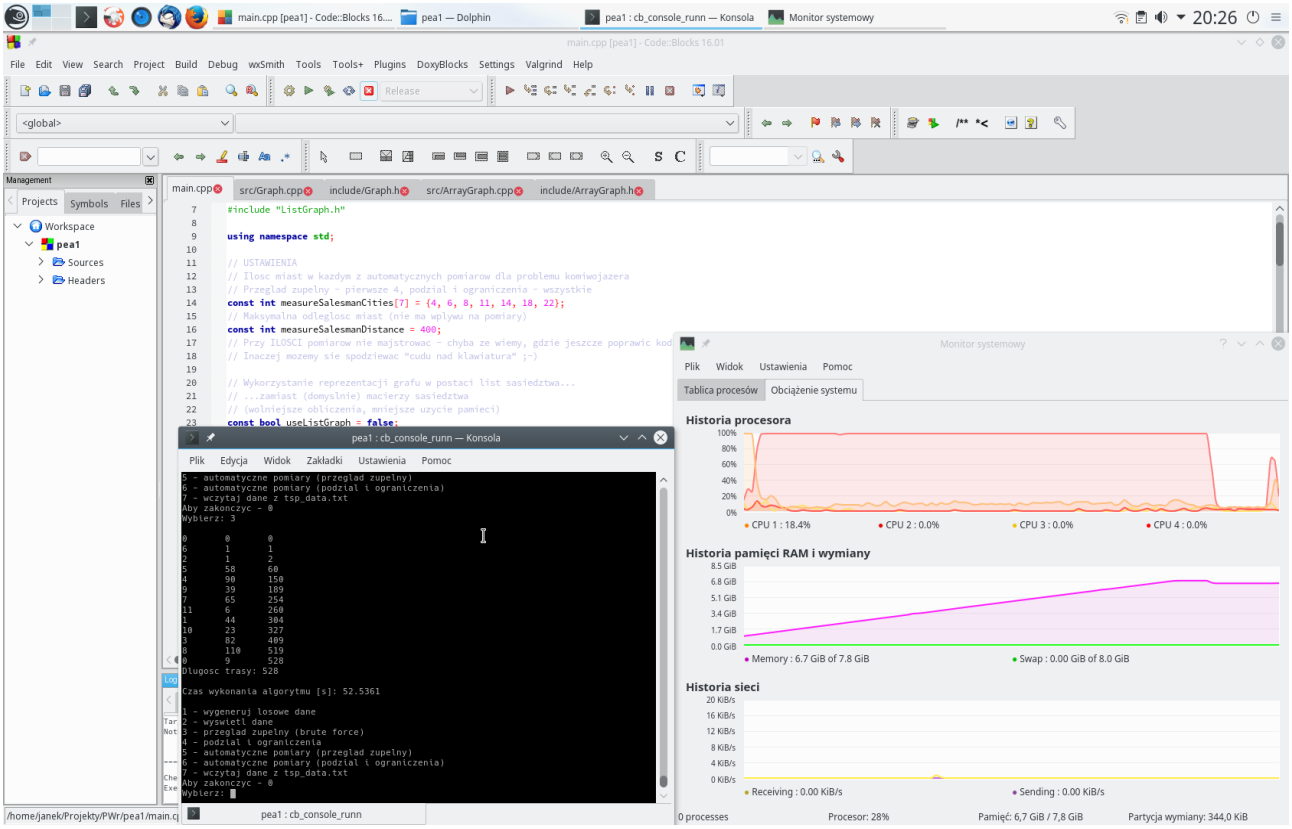
8. Q = {31|(1-2), 31|(1-3-2-5), 34|(1-3-4-5), 37|(1-3-2-4), 38|(1-4-3), 39|(1-3-5), 42|(1-5),
43|(1-3-4-2), 45|(1-4-2), 48|(1-4-5-3)} → v = {1-2}

v == {1-2}
bound(v) == 31 > 30 →break;

3. Opis implementacji algorytmu

Opisany wyżej algorytm zaimplementowałem w języku C++. Graf jest reprezentowany za pomocą własnej struktury danych (stworzonej jeszcze przy pisaniu projektów ze SDiZO): klasy abstrakcyjnej *Graph*, posiadającej dwie implementacje: *ArrayGraph* jako macierz sąsiedztwa i *ListGraph* jako listy sąsiedztwa. Jako kolejkę wykorzystałem adapter std::*priority_queue* z STL, pracujący na kontenerze *std::vector*. Do reprezentacji węzłów drzewa przestrzeni stanów (częściowych rozwiązań) wykorzystałem własną umowną strukturę danych: kontener *std::vector*, którego pierwszym elementem jest wartość jego dolnego ograniczenia, a kolejnymi elementami wierzchołki należące do danego częściowego rozwiązania. Kompletna trasa jest reprezentowana jako *std::vector*, którego kolejnymi elementami są znajdujące się na niej wierzchołki.

4. Środowisko pracy i sposób prowadzenia pomiarów



Ilustracja 2: Wynik przeglądu zupełnego dla 12 miast

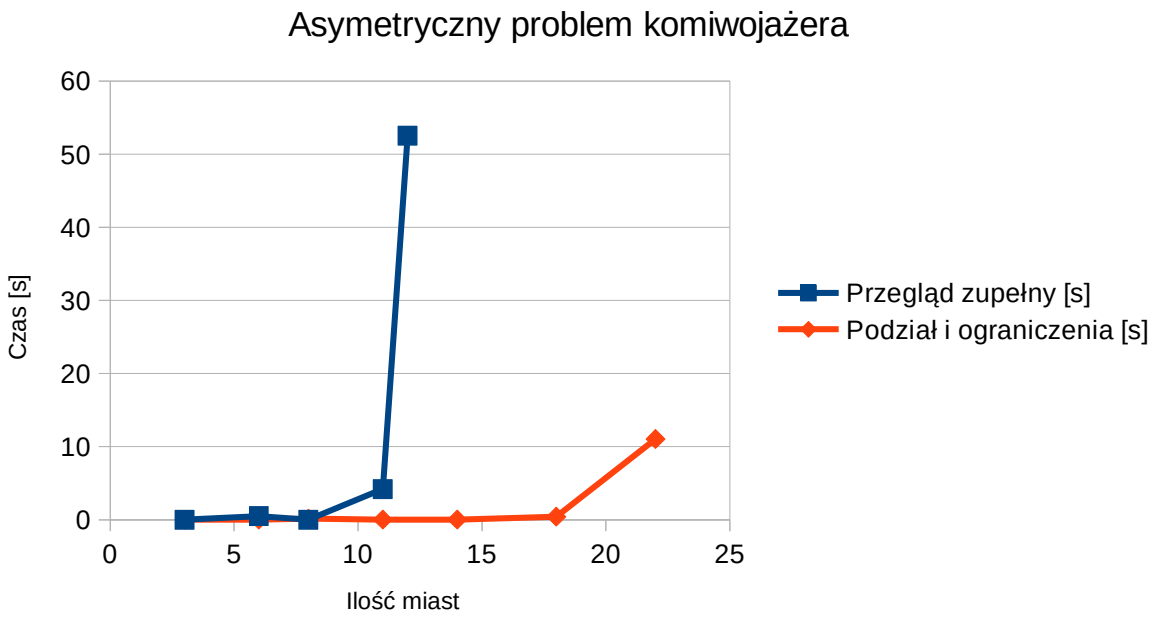
Przy projektowaniu programu wykorzystanego do przeprowadzenia eksperymentu przyjąłem następujące założenia:

- struktury wykorzystywane do reprezentacji danych są alokowane dynamicznie, a struktury pomocnicze (np. zmienne do komunikacji z użytkownikiem) statycznie
- do alokacji i zwalniania pamięci wykorzystuję funkcje *new* i *delete*
- w macierzy i listach sąsiedztwa wierzchołki oraz krawędzie grafu są reprezentowane jako 32-bitowe liczby naturalne (*unsigned*)
- wszystkie pomiary są powtarzane 100 razy, a ich wyniki uśredniane
- pomiary dla algorytmu wykorzystującego przegląd zupełny wykonywane są na 4 ilościach miast: 4, 6, 8, 11 – wyższe wartości nie mają sensu ze względu na długi czas wykonania i zbyt duże wymagania pamięciowe (jeden dodatkowy pomiar dla 12, ostatniego rozmiaru problemu, który na moim komputerze mieści się w pamięci RAM – zob. ilustrację wyżej)

- pomiary dla algorytmu wykorzystującego metodą podziału i ograniczeń wykonywane są na 7 ilościach miast: 4, 6, 8, 11, 14, 18, 22
- program powinien być wieloplatformowy, z tego względu do pomiarów czasu wykorzystuję napisaną przez siebie klasę *Stopwatch*, bazującą na niezależnej od żadnego systemu operacyjnego bibliotece *ctime*

Pomiary przeprowadziłem na komputerze stacjonarnym zbudowanym z wykorzystaniem płyty głównej MSI 870-C45, wyposażonym w czterordzeniowy procesor AMD Phenom II X4 955 BE po overclockingu pracujący z częstotliwością bazową 3.6 GHz i 8 GB pamięci RAM DDR3 pracującej z częstotliwością 1333 MHz w trybie Dual Channel, pod kontrolą 64-bitowego systemu operacyjnego openSUSE Leap 42.3 ze środowiskiem graficznym KDE. Pracowałem w zintegrowanym środowisku programistycznym (IDE) Code::Blocks 16.01, wykorzystując kompilator G++ 4.8 z ustawioną pełną optymalizacją pod kątem szybkości (flaga *-O3*). Przy pomiarach wykorzystałem reprezentację grafu w postaci macierzy sąsiedztwa.

5. Wyniki



Ilość miast	Przegląd zupełny [s]	Podział i ograniczenia [s]
3	0,00000623	0,00001183
6	0,50833200	0,02339070
8	0,01984770	0,17248000
11	4,20448000	0,03775410
12	52,5361000	–
14	–	0,02233150
18	–	0,44274300
22	–	11,0570000

6. Wnioski

Czas wykonania algorytmu przeglądu zupełnego jest stosunkowo duży nawet dla niewielkich rozmiarów problemu, natomiast przy określonym rozmiarze dla każdej instancji problemu jest stały. Czas wykonania algorytmu opartego na metodzie podziału i ograniczeń dla tych samych rozmiarów problemu jest wyraźnie mniejszy, ale bardzo mocno zależy od konkretnej instancji problemu – im większy rozmiar, tym więcej instancji potrzebuje na wykonanie algorytmu czasu dużo dłuższego od średniej, ale jednocześnie zwykle dużo mniejszego od czasu potrzebnego na wykonanie przeglądu zupełnego. Pewne nieścisłości (dla 8 miast średni czas wykonywania przeglądu zupełnego jest niższy niż sąsiednie, a metody podziału i ograniczeń – wyższy) wynikają prawdopodobnie ze specyfiki środowiska pomiarowego: praca pod kontrolą systemu operacyjnego, a więc możliwy wpływ działania innych uruchomionych w systemie procesów na wyniki pomiarów).

Wrocław, 13.11.2017