POLITECHNIKA WROCŁAWSKA

PROJEKTOWANIE EFEKTYWNYCH ALGORYTMÓW
PROJEKT, WT 17:05
K00-58d

PROBLEM KOMIWOJAŻERA

BRUTE FORCE
BRANCH AND BOUND - LOW COST

Katarzyna Hajduk, 259189
ITE 2020, W4N

Prowadzący:
Dr inż. Jarosław Mierzwa

15 listopada 2022

1. Wstęp teoretyczny

Zadanym problemem jest zaimplementowanie oraz dokonanie analizy efektywności algorytmu przeglądu zupełnego oraz podziału i ograniczeń (*B&B*) dla asymetrycznego problemu komiwojażera (*ATSP*). Problem komiwojażera jest to zagadnienie optymalizacyjne polegające na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie. Cykl Hamiltona jest to taki cykl w grafie, w którym każdy wierzchołek grafu został odwiedzony dokładnie raz w wyłączeniem pierwszego wierzchołka.

2. Założenia

Podczas realizacji zadania należy przyjąć następujące założenia:

- używane struktury danych powinny być alokowane dynamicznie (w zależności od aktualnego rozmiaru problemu),
- program powinien umożliwić weryfikację poprawności działania algorytmu (wczytanie danych wejściowych z pliku tekstowego),
- po zaimplementowaniu i sprawdzeniu poprawności działania algorytmu należy dokonać pomiaru czasu jego działania w zależności od rozmiaru problemu N (badania należy wykonać dla minimum 7 różnych reprezentatywnych wartości N),
- dla każdej wartości N należy wygenerować po 100 losowych instancji problemu (w sprawozdaniu należy umieścić tylko wyniki uśrednione),
- implementacje algorytmów powinny być zgodne z obiektowym paradygmatem programowania,
- używanie „okienek” nie jest konieczne i nie wpływa na ocenę (wystarczy wersja konsolowa),
- kod źródłowy powinien być komentowany,
- warto pamiętać o dużych różnicach w wynikach testów czasowych pomiędzy wersjami *Debug* i *Release* (testy trzeba przeprowadzić w wersji *Release*).

3. Implementacja algorytmu przeglądu zupełnego

Algorytm przeglądu zupełnego (ang. *brute force*) polega na rozpatrzeniu wszystkich możliwości, a następnie wybranie tej najbardziej optymalnej. Oznacza to, że musimy znaleźć wszystkie permutacje drogi. Główną trudnością związaną z poszukiwaniem rozwiązania dla tego problemu komiwojażera jest bardzo duża liczba danych do analizy.

Złożoność obliczeniowa algorytmu dla przeglądu zupełnego wynosi: O(N!), gdzie N – liczba wierzchołków w grafie, a więc liczba rozpatrywanych miast.

W implementacji algorytmu używam biblioteki *algorithm*, aby uzyskać wszystkie permutacje. Biorę po kolei miasta „w parach”, a następnie sumuję wszystkie dystanse.

Do implementacji wykorzystuję następujące struktury:

- Tablica 2-wymiarowa, która przechowuję reprezentację grafu,
- Tablice 1-wymiarowe, których używam do wyznaczenia kolejnych permutacji oraz przechowującą ścieżkę.

4. Implementacja algorytmu podziału i ograniczeń

Metoda podziału i ograniczeń, opiera się na przeszukiwaniu drzewa reprezentującego przestrzeń rozwiązań problemu metodą w głąb, wszerz, lub najmniejszym kosztem. Własnością algorytmu jest ograniczanie ilości przeglądanych rozwiązań, co w konsekwencji redukuje ilości czasu potrzebnego do wyznaczenia najlepszej wartości.

Algorytm w wersji *low cost* zwraca zawsze wierzchołek z najniższym kosztem. Ograniczenie dolne odbywa się poprzez wyznaczenie najmniejszych wartości dla każdego rzędu oraz każdej kolumny oraz redukcję macierzy o te koszty tak, że w każdym rzędzie i każdej kolumnie musi być przynajmniej jedno zero. Wartością dolnego ograniczenia jest suma najmniejszych wartości z każdego wiersza oraz każdej kolumny.

Do implementacji wykorzystuję następujące struktury:

- Wektor 2-wymiarowy, w którym przechowuję reprezentację grafu,
- Wektory 3-, 2- i 1- wymiarowe, których używam jako zmiennych pomocniczych. Przechowuję w nich informację o rozpatrywanych matrycach, sekwencjach oraz koszcie.

Przykład:

Szukamy minimum dla każdego rzędu i odejmujemy tą wartość, tak, aby w każdym rzędzie było przynajmniej jedno 0. Analogicznie postępujemy dla kolumn.

	0	1	2	3	**Min**
0	-1	4	12	7	**4**
1	5	-1	8	18	**5**
2	11	2	-1	6	**2**
3	9	3	2	-1	**2**

	0	1	2	3
0	-1	0	8	3
1	0	-1	3	13
2	9	0	-1	4
3	7	1	0	-1
Min	**0**	**0**	**0**	**3**

	0	1	2	3
0	-1	0	8	0
1	0	-1	3	10
2	9	0	-1	1
3	7	1	0	-1

Wyliczamy dolne ograniczenie: LB = 4 + 5 + 2 + 2 + 3 = 16.

Następnym krokiem jest obliczenie dolnych ograniczeń dla następników. Dla połączenia (0,1) dla wszystkich wartości w wierszu zerowym i kolumnie pierwszej oraz dla punktu (1,0) ustawiamy nieskończoność.

	0	1	2	3
0	-1	-1	-1	-1
1	-1	-1	3	10
2	9	-1	-1	1
3	7	-1	0	-1

	0	1	2	3
0	-1	-1	-1	-1
1	-1	-1	0	7
2	1	-1	-1	0
3	0	-1	0	-1

cost[0] = 3 + 1 + 7 = 11

Cost = cost[0] + matrix[0][1] + LB = 11 + 0 + 16 = 27.

Analogicznie liczymy dla pozostałych przypadków.

5. Implementacja czasu

Pomiar czasu zmierzyłam za pomocą QueryPerformanceCounter(). Zwracany stan licznika jest dzielony przez częstotliwość. Tak otrzymujemy pomiar w mikrosekundach.

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

Rys 1: Fragment kodu odpowiedzialny za pomiar czasu.

6. Plan eksperymentu

6.1. Rozmiar używanych struktur

W metodzie Brute Force odległości pomiędzy poszczególnymi punktami są przechowywane w tablicy alokowanej dynamicznie o rozmiarach NxN. Optymalna ścieżka jest przechowywana w tablicy o wielkości N.

W metodzie Branch and Bound odległości pomiędzy poszczególnymi punktami są przechowywane w wektorze alokowanym dynamicznie o rozmiarze NxN. Tymczasowe matryce są przechowywane w wektorze 3-wymiarowym, a optymalna ścieżka jest przechowywana w wektorze o wielkości NxN.

Rozpatrywana wielość N dla poszczególnych algorytmów:

- Algorytm przeglądu zupełnego – N = [6; 14]
- Algorytm podziału i ograniczeń – N = [6; 15]

Dla algorytmu podziału i ograniczeń zostało znalezione maksymalne N, z którym algorytm poradził sobie w przeciągu przyjętego czasu 2 minut i wynosiło N = 25.

6.2. Sposób generowania danych

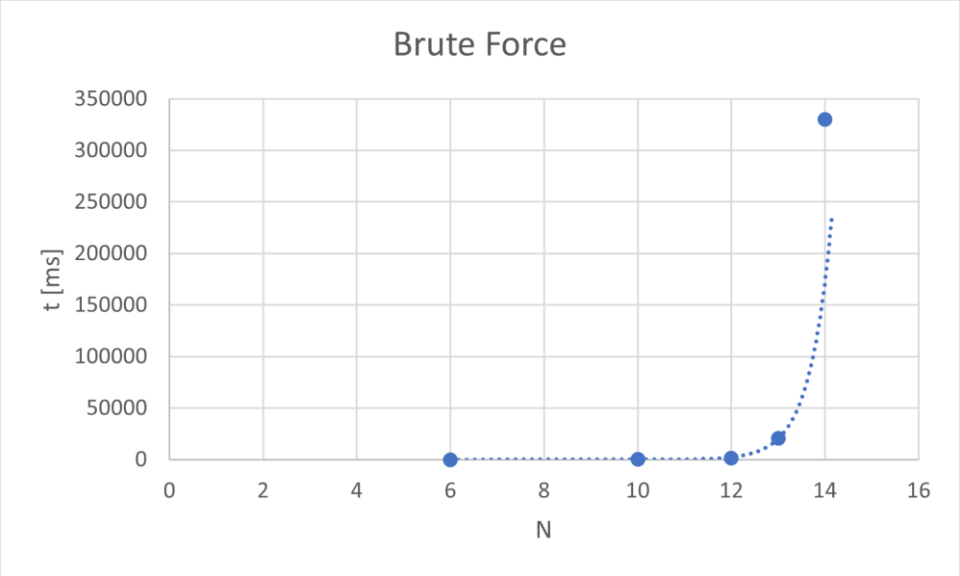
Generowanie matrycy jest tworzone na podstawie danych przekazanych od użytkownika – należy podać rozmiar oraz zakres zbioru wartości. Następnie dane są losowane przy pomocy funkcji rand(). Dane znajdujące się po przekątnej przyjmują wartość -1.

```
for (int i = 0; i < size; i++){
        for (int j = 0; j < size; j++){
            if(i != j) matrix[i][j] = rand()%(y-x+1)+x;
            else matrix[i][j] = -1;
        }
    }
}
```

Rys 2: Fragment kodu odpowiedzialny za generowanie matrycy.

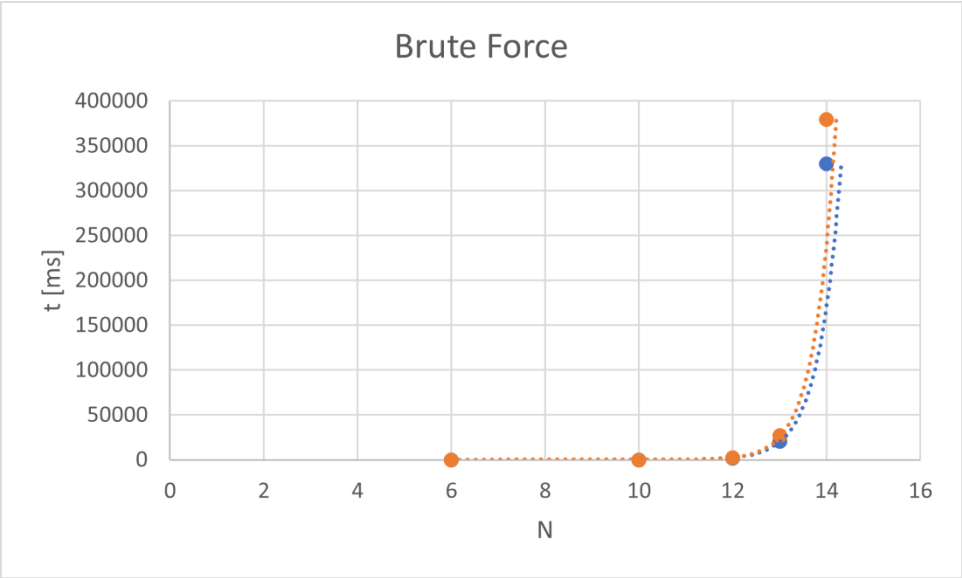
7. Wyniki

Uśrednione wyniki z prób zostały przedstawione na wykresie i w tabelce poniżej:

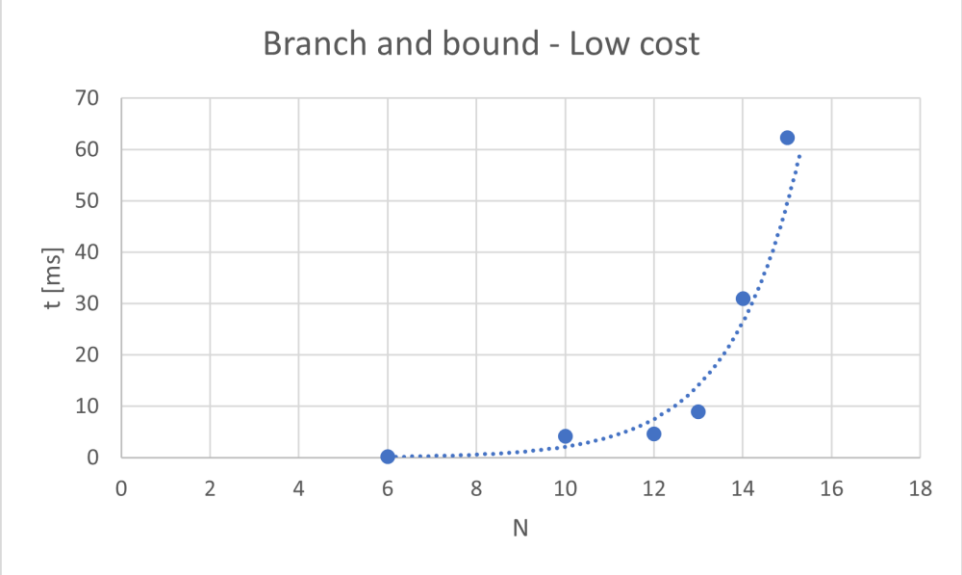


Rys 3: Wykres przedstawiający uśredniony czas [s] wykonania algorytmu Brute Force dla ilości miast N.

N	6	10	12	13	14
t [ms]	0.0103	15.8	1610	20700	329800

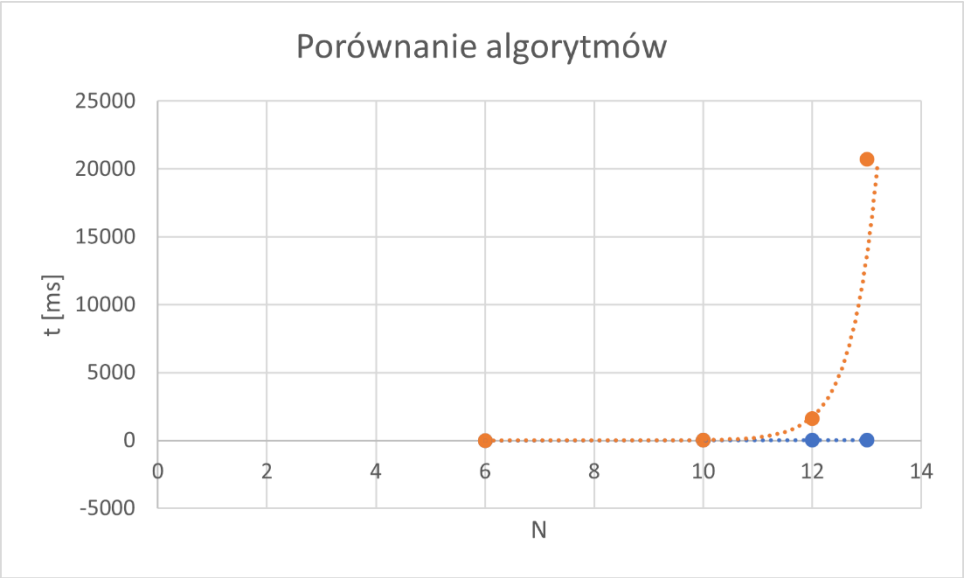


Rys 4: Wykres przedstawiający uśredniony czas [s] wykonania algorytmu Brute Force dla ilości miast N (niebieska linia), w porównaniu do przewidywanego czasu dla złożoności O(N!) (pomarańczowa linia).



Rys 5: Wykres przedstawiający uśredniony czas [s] wykonania algorytmu Branch and Bound dla ilości miast N.

N	6	10	12	13	14	15
t [ms]	2.7	3.38	3.67	3.9	4.3	6.72



Rys 6: Porównanie algorytmów Branch and Bound (niebieska linia) i Brute Force (pomarańczowa linia).

8. Wnioski

Porównując uzyskane wyniki z oszacowanymi z przewidywanymi złożonościami dla każdej metod można uznać, że algorytmy zostały zaprojektowane poprawnie.

Algorytm przeglądu zupełnego wykazał złożoność obliczeniową równą O(N!). Metoda ta nadaje się jedynie do rozwiązywania instancji problemu o bardzo małych rozmiarach oraz dla tych przypadków jest najszybsza.

Algorytm ograniczeń i podziału w najgorszym wypadku również ma złożoność O(N!). Jednakże, dzięki ograniczaniu i rozgałęzianiu czas wykonywania programu algorytmem Branch and Bound z przeszukiwaniem *low cost* jest lepszy niż Brute Force. Niestety dużym minusem algorytmu BnB w moim wykonaniu jest duże zużycie pamięci, przez co dla bardziej złożonych przypadkach może się to okazać problemem.