Politechnika Wrocławska

Wydział Informatyki i Telekomunikacji

Projektowanie efektywnych algorytmów

Projekt nr 1 - analiza efektywności algorytmu przeglądu zupełnego dla asymetrycznego problemu komiwojażera (ATSP)

semestr zimowy 2023/2024

Autor:
Eryk Mika 264451

Prowadzący:
Dr inż. Marcin Łopuszyński

Spis treści

1. Wstęp teoretyczny

Asymetryczny problem komiwojażera (ang. Asymmetric Traveling Salesman Problem, ATSP) to jedno z najbardziej znanych i intensywnie badanych zagadnień w dziedzinie optymalizacji kombinatorycznej. W kontekście ATSP zakłada się, że mamy daną listę miast oraz odległości między każdą parą miast, ale odległości te nie muszą być takie same w obie strony. Celem jest znalezienie najkrótszej trasy, która przebiega przez każde miasto dokładnie raz i wraca do miasta początkowego.

Badanie omawianego problemu można rozważać za pomocą reprezentacji grafowej. W języku grafów, miasta są punktami (wierzchołkami), a drogi pomiędzy nimi stanowią połączenia (krawędzie) z określonymi wartościami – na przykład wspominanymi odległościami. Trasa komiwojażera to cykl Hamiltona, czyli ścieżka w grafie, w której wszystkie wierzchołki występują tylko raz poza wierzchołkiem początkowym, który jest także wierzchołkiem końcowym.

Znalezienie cyklu Hamiltona, który jest „najlepszy" (najkrótszy) jest zadaniem bardzo trudnym obliczeniowo. W skrajnym przypadku będziemy mieli do czynienia z grafem pełnym, czyli takim, w którym każdy wierzchołek jest połączony z każdym. Oznacza to, że przy wykorzystaniu metody przeglądu zupełnego (ang. Brute Force), w której sprawdzamy wszystkie możliwości, będziemy musieli sprawdzić aż $(\mathrm{n}-1)!$ możliwych ścieżek w grafie dla grafu o n wierzchołkach. Jest to najprostsze podejście, ale staje się ono niepraktyczne dla większych instancji problemu, gdyż liczba permutacji rośnie bardzo szybko wraz ze wzrostem n – algorytm ma złożoność czasową $O(n!)$. Jest to problem NP-trudny, czyli nie jest możliwe jego rozwiązanie ze złożonością wielomianową.

2. Opis implementacji algorytmu

W celu analizy efektywności algorytmu przeglądu zupełnego dla rozważanego problemu został napisany program w języku C++ z wykorzystaniem obiektowego paradygmatu programowania. Najistotniejszymi komponentami aplikacji są klasy *Graph* oraz *PermutationArray*, których pola (struktury danych) oraz metody są odpowiedzialne za realizację algorytmu. Wiele istotnych kwestii związanych z implementacją zostało wyjaśnionych w komentarzach w plikach źródłowych.

2.1. Klasa Graph

Klasa Graph jest główną klasą programu, która jest odpowiedzialna za przechowywanie struktury i metod grafu, na którym wykonywany jest algorytm przeglądu zupełnego. Pola prywatne klasy – wskaźnikowe *matrix* oraz *size* są użyte do przechowywania długości krawędzi w postaci macierzy kwadratowej *matrix* stopnia *size*. Oba pola przechowują liczby stałoprzecinkowe typu *int*. W macierzy komórka o współrzędnych *i*, *j* zawiera odległość pomiędzy wierzchołkami *i* i *j.* Zaimplementowano konstruktory (domyślny, generujący losową instancję problemu

o rozmiarze N, wczytujący instancję z pliku tekstowego), destruktor zwalniający dynamicznie alokowaną pamięć, przeładowany operator przypisania oraz metodę wypisującą graf (macierz) na ekran. Konstruktor generujący losową instancję problemu wykorzystuje funkcję *rand()* z biblioteki *<random>* do generowania liczb pseudolosowych – długości krawędzi po uprzednim zainicjalizowaniu generatora za pomocą wywołania funkcji *srand(time(NULL))* i wykorzystaniu biblioteki *<time.h>*. Najważniejszą metodą klasy jest *timeBruteForceATSP(),* która zwraca czas (w mikrosekundach) potrzebny na rozwiązanie problemu ATSP. W metodzie tej, korzystając z kolejnych generowanych permutacji ciągu *n-1* wierzchołków (poza pierwszym – wszystkie cykle Hamiltona w grafie pełnym dla jednego wierzchołka są równoważne ze wszystkimi cyklami Hamiltona dla innego wierzchołka w tym grafie), obliczamy sumaryczny koszt dla danej ścieżki przechodzącej przez te wierzchołki. Jeżeli okaże się, że obecna ścieżka jest lepsza od dotychczas najlepszego rozwiązania, zapisujemy nowy koszt oraz ścieżkę (Rysunek 2.1). Oprócz znalezienia czasu rozwiązania, na ekran wypisujemy także znaleziony koszt oraz ścieżkę.

Rysunek 2.1 Główna część algorytmu znajdowania najlepszego cyklu Hamiltona (rozwiązania problemu ATSP)

```
// Korzystajac z metody generujacej kolejne permutacje, przechodzimy po wszystkich w petli for
for(int p=0; p < PermutationArray::factorial(permutatedElements); p++)
{
    // Obecny koszt - zaczynamy od krawedzi laczacej wierzcholek startowy oraz pierwszy wierzcholek z permutacji n-1 pozostalych
    unsigned int current_cost = matrix[source][permutation[0]];

    // Przechodzimy po grafie wg obecnej permutacji dodajac wagi krawedzi do kosztu
    for(int i=0; i<permutatedElements-1; i++)
    {
        // Dla danego i dodajemy dlugosc krawedzi laczacej wierzcholki i oraz i+1.
        current_cost += matrix[permutation[i]][permutation[i+1]];

        // Jezeli gorzej niz dotychczasowe minimum - przerywamy i kontynnujemy od kolejnej permutacji
        if(current_cost >= min_cost)
            break;
    }

    // Konczymy cykl Hamiltona
    current_cost += matrix[permutation[permutatedElements-1]][source];

    // Jezeli mamy nowe najlepsze rozwiazanie - przypisujemy koszt i zapisujemy sciezke
    if(current_cost < min_cost)
    {
        min_cost = current_cost;

        for(int i=0; i<permutatedElements; i++)
        {
            min_solution[i] = permutation[i];
        }
    }
    // Metoda generujaca kolejne leksykograficzne permutacje wewnetrznej tablicy obiektu permutation
    permutation.nextPermutation();
}
```

Źródło: opracowanie własne

2.2. Klasa PermutationArray

Klasa ta odpowiada za przechowywanie tablicy reprezentującej ciąg wierzchołków, dla którego generowane są następne leksykograficzne permutacje. Umożliwia to wyszukiwanie kolejnych ciągów wierzchołków – ścieżek, a tym samym cyklów Hamiltona, w głównej metodzie klasy *Graph*. Zaimplementowano konstruktory, w tym tworzący permutowany ciąg wierzchołków 1 do n. Stworzono pomocnicze metody statyczne – *factorial(int n)*, która służy do obliczenia silni – liczby możliwych cyklów Hamiltona dla grafu, co jest wykorzystane w metodzie znajdującej rozwiązanie problemu ATSP (Rysunek 2.1), oraz metodę *swap()* używaną do zamiany elementów

wewnętrznej tablicy obiektu klasy podczas generowania permutacji. Tablica *arr* o rozmiarze *size* jest alokowana dynamicznie podobnie jak w klasie *Graph*.

Najważniejsza metoda klasy – *nextPermutation()* jest użyta do generowania kolejnych permutacji według porządku leksykograficznego. Odbywa się to według następującego algorytmu, który jest także opisany w komentarzach w pliku źródłowym *PermutationArray.cpp*:

1. Przechodząc po elementach tablicy od jej prawej strony, znajdujemy pierwszy element, który nie jest większy od poprzedniego elementu,
2. Ponownie przechodzimy od prawej strony tablicy i znajdujemy pierwszy najmniejszy element większy od elementu wyznaczonego w kroku 1,
3. Zamieniamy elementy określone w krokach 1 i 2 miejscami,
4. Odwracamy część tablicy począwszy od elementu następującego po elemencie o indeksie z kroku 1 do końca tablicy. Otrzymujemy następną szukaną permutację.

3. Sposób przeprowadzenia eksperymentu

W celu realizacji badania – eksperymentu zostały wykorzystane wcześniej opisane klasy. W metodzie klasy *Graph - timeBruteForceATSP()* przeprowadzamy pomiar czasu potrzebnego na znalezienie optymalnego rozwiązania, korzystając z funkcjonalności biblioteki *<chrono>* i zawartej w niej klasy *steady_clock*, która reprezentuje zegar monotoniczny, dla którego gwarantowane jest, że różnica czasu (przykładowo przekonwertowanego do mikrosekund – tak jak w programie) będzie większa od zera dla dwóch momentów czasu badania, z których drugi występuje później niż pierwszy. W trakcie implementacji okazało się, że jest to lepsze rozwiązanie od użycia klasy *high_resolution_clock*, która nie zawsze pozwalała na uzyskanie poprawnych rezultatów. Różnica czasów pobranych z zegara przed i po wykonaniu algorytmu jest zwracana z metody. Nie uwzględniamy jednak czasu wypisywania znalezionego kosztu, ciągu wierzchołków oraz czasu działania algorytmu.

W głównym programie, z którego korzysta użytkownik, możliwe jest wywołanie metody dla pojedynczej instancji problemu o rozmiarze N, która może być wylosowana lub wczytana z pliku, oraz przeprowadzenie zbiorczego badania dla 100 losowych instancji problemu o rozmiarze N, dzięki czemu możliwe jest znalezienie średniego czasu działania algorytmu jako iloraz sumy czasów poszczególnych realizacji algorytmu i ich ilości. Opcja ta została wykorzystana do przeprowadzenia analizy efektywności algorytmu rozwiązującego problem ATSP, która jest przedmiotem tego projektu. Wybrano 7 reprezentatywnych wartości N.

Badania zostały przeprowadzone na laptopie z systemem operacyjnym Windows 10 Home, procesorem Intel Core i7-8750H, 24 GB pamięci RAM i dyskiem SSD.

4. Wyniki przeprowadzonego eksperymentu

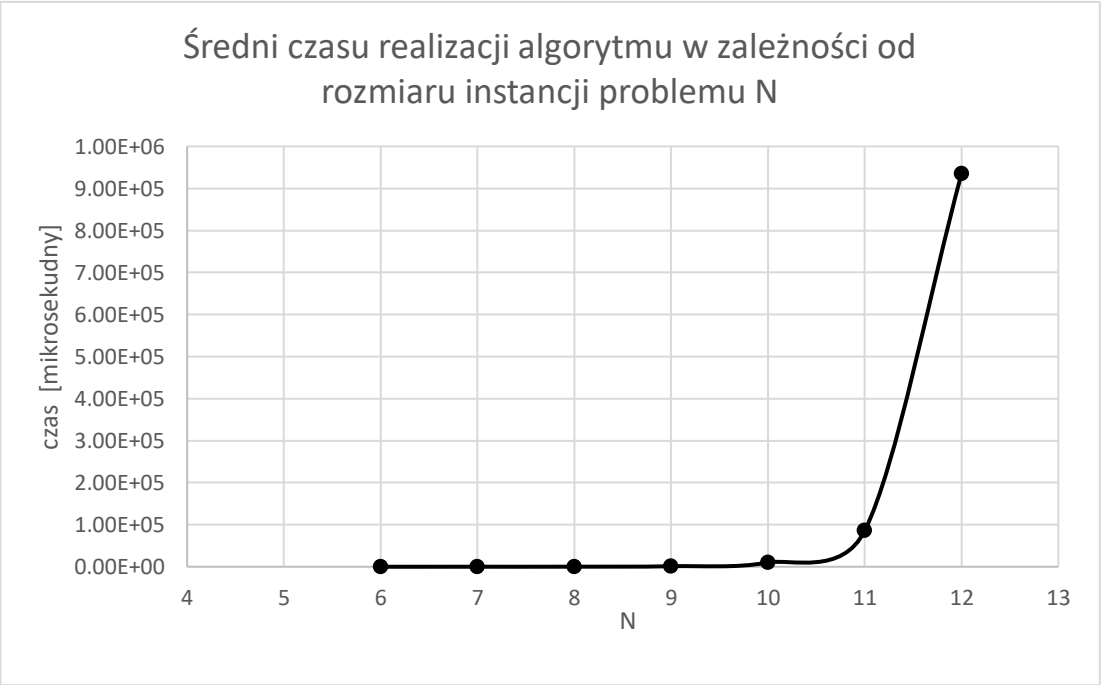
Analizę efektywności działania algorytmu przeglądu zupełnego dla problemu ATSP przeprowadzono dla instancji problemów o rozmiarach *N* od 6 do 12. Wyniki eksperymentu w postaci średnich czasów wykonywania algorytmu oraz wartości mediany przedstawiono za pomocą tabeli oraz wykresu. Można zauważyć, że średnie wartości są zbliżone do wartości mediany (Tabela 4.1).

Tabela 4.1 Rezultaty analizy efektywności algorytmu

N	średnia (w mikrosekundach)	mediana (w mikrosekundach)
6	5	5
7	26	26
8	165	164
9	1270	1272
10	10072	9991
11	85686	84974
12	935180	926078

Źródło: opracowanie własne

Wykres 4.1 Zależność średniego czasu wykonywania algorytmu od rozmiaru problemu



Źródło: opracowanie własne

5. Wnioski

Można łatwo wywnioskować, że algorytm przeglądu zupełnego dla rozwiązywania asymetrycznego problemu komiwojażera jest algorytmem stosunkowo łatwym do implementacji i zrozumienia, jednak zdecydowanie nieefektywnym pod względem obliczeniowym. Ze względu na złożoność algorytmu typu n-silnia czas potrzebny na rozwiązanie problemu rośnie bardzo szybko wraz ze wzrostem rozmiaru problemu *N*, co powoduje, że dla instancji o rozmiarze większym od 10 metoda ta staje się zupełnie niepraktyczna. Udało się poprawnie zaimplementować analizowany algorytm oraz zbadać jego złożoność obliczeniową.

6. Bibliografia

[1] Problem komiwojażera - https://eduinf.waw.pl/inf/alg/001_search/0140.php

[2] Znajdowanie permutacji - https://www.nayuki.io/page/next-lexicographical-permutation-algorithm

[3] std::chrono - https://en.cppreference.com/w/cpp/chrono