Projektowanie Efektywnych Algorytmów

Zadanie projektowe nr 1A

Implementacja i analiza efektywności algorytmu przeglądu zupełnego dla asymetrycznego problemu komiwojażera (ATSP).

Wtorek, 15:15 TNP

Autor:

Michał Lewandowski #264458

1. Wstęp teoretyczny

Problem komiwojażera to zagadnienie optymalizacyjne z teorii grafów polegające na odnalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Dla łatwiejszego zobrazowania możemy przedstawić to w ten sposób: przedstawiciel handlowy ma listę pewnej liczby miast, które musi odwiedzić a następnie powrócić do miasta początkowego. Problem polega na tym, aby tak zaplanował swoją podróż aby przejazdy między miastami kosztowały go jak najmniej (np. aby odległości były jak najkrótsze lub jak najmniej zapłacił za paliwo). Co ważne: zakładamy, że graf jest asymetryczny, tzn. waga tej samej krawędzi (koszt tej samej trasy) jest różna w zależności w którą stronę chcemy przez nią przejść.

1.1. Metoda przeglądu zupełnego

Metoda przeglądu zupełnego polega na wyznaczeniu wszystkich możliwych rozwiązań danego problemu. Dalej wyznaczane są dla nich wartości funkcji celu oraz wybierane jest rozwiązanie o ekstremalnej wartości funkcji celu, odpowiednio najniższej lub najwyższej. Zaletą metody jest stosunkowa łatwość implementacji, wadą jest wysoka złożoność obliczeniowa. W kontekście problemu komiwojażera, użycie tej metody sprowadza się do sprawdzenia wszystkich możliwych dróg, zgodnych z założeniami oraz wyznaczenia tej, której sumaryczna wartość ścieżek jest najmniejsza.

1.2. Złożoność obliczeniowa

Złożoność obliczeniowa to ilość zasobów komputerowych, potrzebnych do jego wykonania. Skupimy się na złożoności czasowej, czyli na ilości czasu potrzebnego do wykonania zadania, która zostanie wyrażona jako funkcja ilości danych.

1.3. Złożoność przeglądu zupełnego

Generowanie wszystkich możliwych ścieżek to obliczenie wszystkich możliwych permutacji, których jest aż (n-1)!. Z tego powodu złożoność czasowa wynosi O(n!).

2. Opis implementacji algorytmu

Klasa Brute_force

Konstruktor klasy Brute_force do inicjalizacji wymaga grafu w formie macierzy sąsiedztwa dwuwymiarowej, alokowanej dynamicznie.

Wartości grafu oznaczone indeksem [i][j] oznaczają odpowiednio wagę krawędzi prowadzącej od wierzchołka i do j, pod warunkiem że są to różne wierzchołki (zakładamy brak pętli).

Strukturą danych użytą do przechowywania grafu jest Vector.

Vector jest dynamiczną tablicą która automatycznie alokuję, relokuję oraz zwalnia pamięć, w dodatku jest bardzo wydajną strukturą danych.

```
Brute_force::Brute_force(std::vector<std::vector<int>> graph_matrix) {

    this->graph = graph_matrix;
    this->num_of_vertices = graph_matrix.size();
    this->lowest_cost = INT_MAX;
    this->current_cost = 0;

    this->lowest_path = std::vector<int>( n: this->num_of_vertices, value: 0);
    prepare_permutations();
}
```

Rys 1. Konstruktor klasy Brute_force

Zmienna num_of_vertices przechowuje rozmiar grafu (ilość wierzchołków), zmienne lowest_cost i current_cost służą do porównywania kosztu drogi aktualnej iteracji z obecnie znalezionym najmniej kosztowną drogą natomiast zmienna lowest_path przechowuje obecnie najmniej kosztowną permutacje.

Metoda prepare_permuations() przygotowuje wszystkie możliwe permutacje dla grafu.

```cpp
// Wykonuje przeglad zupelny
void Brute_force::perform_brute_force() {

    do {
        current_cost = 0;
        current_cost += graph[0][permutations[0] + 1];
        for (int i = 0; i < num_of_vertices - 2; ++i) {
            current_cost += graph[(permutations[i] + 1)][(permutations[i + 1] + 1)];
        }
        current_cost += graph[permutations.back() + 1][0];

        if (current_cost < lowest_cost) {
            this->lowest_cost = current_cost;
            this->lowest_path = permutations;



        }

    } while (std::next_permutation( first: permutations.begin(),  last: permutations.end()));
}
```

Rys 2. Metoda perform_brute_force klasy Brute_force

Metoda perform_brute_force() służy do wykonania algorytmu przeszukania zupełnego na grafie przekazanym w konstruktorze. Korzystając z biblioteki STL oraz funkcji next permutation() do generowania permutacji w każdej iteracji zliczany jest całkowity koszt drogi który jest porównywany z obecnie znalezioną najmniej kosztowną drogą, jeżeli koszt tej drogi jest mniejszy od znalezionej, zostaje on zaktualizowany a permutacja zostaje przypisana do zmiennej lowest_path, algorytm działa dopóki wszystkie permutacje nie zostaną sprawdzone.

Klasa zawiera również metodę show_lowest_path() która odpowiedzialna jest za wyświetlnie w konsoli najmniej kosztownej drogi i jej koszt.

3. Plan eksperymentu

W celu zapewnienia miarodajnej analizy implementacji algorytmu, został zmierzony czas dla grafów 5, 6, 7, 8, 9, 10, 11, 12 i 13 wierzchołkowych gdzie dla każdego rozmiaru grafu zostało przeprowadzonych 100 pomiarów czasowych losowych instancji problemu których wyniki uśredniono.

```
long long Time_measure::test_brute_force_random_matrices(int matrix_size) {
    long long sum = 0;
    std::random_device rd;
    std::mt19937 gen( sd: rd());
    std::uniform_int_distribution<int> random( a: 1,  b: 200);
    std::vector<std::vector<int>>  temp_matrix( n: matrix_size,  value: std::vector<int>( n: matrix_size,  value: 0));
    for (int t = 0; t < num_of_tests; t++) {

        for (int i = 0; i < matrix_size; i++) {
            for (int j = 0; j < matrix_size; j++) {
                if (i == j) {
                    temp_matrix[i][j] = -1;
                } else {
                    temp_matrix[i][j] = random( &: gen);
                }
            }
        }

        Brute_force brute_test( graph_matrix: temp_matrix);
        auto start : time_point<...>  = std::chrono::high_resolution_clock::now();
        brute_test.perform_brute_force();
        auto end : time_point<...>  = std::chrono::high_resolution_clock::now();
        auto duration : duration<...>  = std::chrono::duration_cast<std::chrono::microseconds>( d: end - start);
        std::cout << "Time taken by function: " << duration.count() << " microseconds" << std::endl;
        sum += duration.count();
    }
    long long averageTime = sum / num_of_tests;
    std::cout << "Average time taken to perform brute force " << averageTime << " microseconds.\n" << num_of_tests << " test done\n";
    return averageTime;
}
```

Rys 3. Metoda test_brute_force_random_matrices klasy Time_measure

Grafy wejściowe były generowane jako macierze zawierające losowe liczby z przedziału od 1 do 200 a dane znajdujące się na ich przekątnej zostały przypisane wartości -1.

Do generowania losowych liczb wykorzystałem bibliotekę <random>.

Pomiary czasów zostały przeprowadzone przy pomocy funkcji std::chrono::high_resolution_clock::now() z bilbioteki <chrono>. Czas został zmierzony w mikrosekundach.

Testy odbywały się używając trybu RELEASE środowiska CLion na proceszorze Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz 3.60 GHz z 32,0 GB pamięci RAM o szybkości 2133 MHz i systemie operacyjnym Windows 10 Education N 21H2

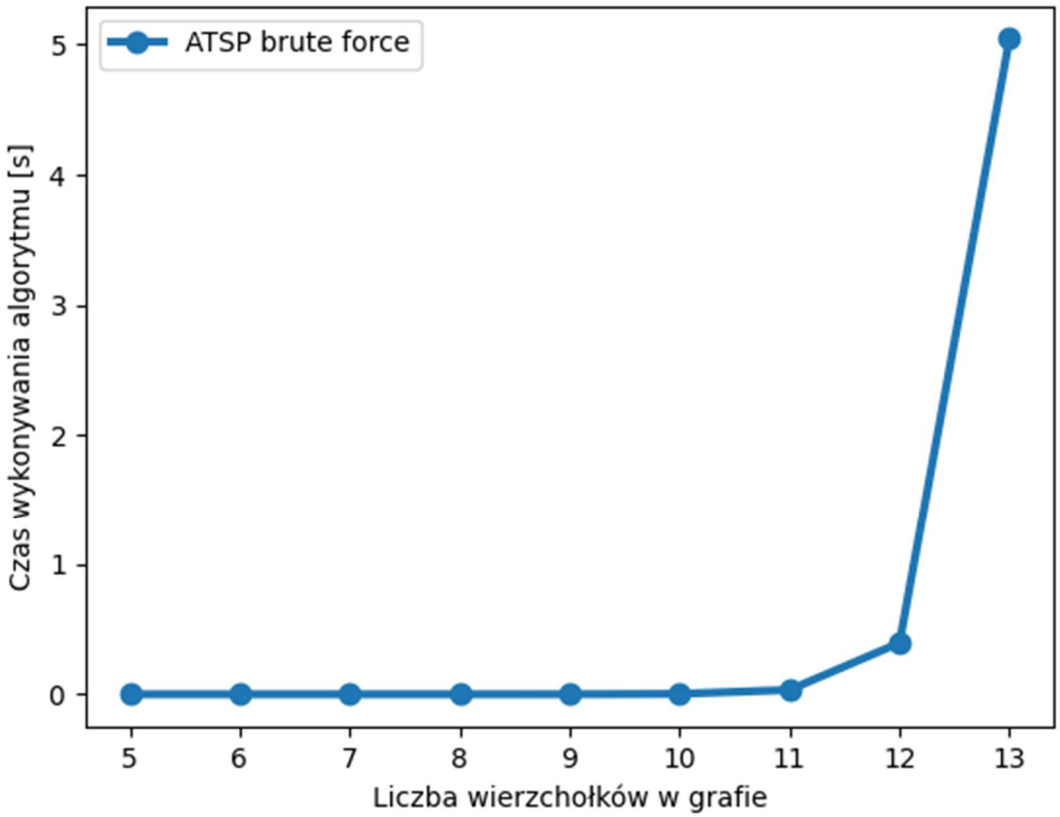
4. Wyniki eksperymentów

Tabela 1. Uśredniony czas wykonania algorytmu dla różnej liczby wierzchołków

Liczba wierzchołków	Czas [μs]	Czas [s]
5	0	0
6	5	5,00E-08
7	411	4,11E-06
8	3624	3,62E-05
9	32185	3,22E-04
10	313415	3,13E-03
11	3362518	3,36E-02
12	39496469	3,95E-01
13	5,05E+08	5,05E+00

Uzyskany uśredniony czas świadczy o prawidłowej implementacji algorytmu i jest bliski oczekiwanemu czasowi dla przeglądu zupełnego. Brak pomiarów dla większej liczby wierzchołków spowodowany jest zbyt długim czasem oczekiwania na wyniki.

Wykres 1. Zależność czasu wykonania od rozmiaru problemu dla przeglądu zupełnego



Na podstawie wygenerowanego wykresu można zaobserwować wyraźny wzrost czasu wykonania algorytmu wraz z rosnącą liczbą wierzchołków. Z powodu ogromnego rozrzutu czasu wykonania dla poszczególnych liczb wierzchołków oś y wykresu została przedstawiona w sekundach i dokładne wartości osi y nie są czytelne.

5. Wnioski

Wyniki pomiarów wykazały spójność między oczekiwaniami teoretycznymi a praktycznymi rezultatami uzyskanymi poprzez implementacje algorytmu, zaimplementowany algorytm przeglądu zupełnego wykazał jego teoretyczną, wykładniczą złożoność czasową O(n!).

Algorytm przeglądu zupełnego zawsze znajduje optymalne rozwiązanie, ponieważ przeszukuje wszystkie możliwe kombinacje tras. Optymalność rozwiązania jest osiągana kosztem czasu wykonania, dla większych instancji problemu komiwojażera, czas wykonania algorytmu przeglądu zupełnego staje się bardzo długi i nieefektywny.