Projektowania Efektywnych Algorytmów
Sprawozdanie nr 1
Problem komiwojażera

Jan Woźniak 234995

1. Wstęp.

Celem projektu było napisanie przynajmniej dwóch algorytmów rozwiązujących problem komiwojażera. Do wyboru był algorytm BnB, Brute Force i Helda Karpa. Projekt został wykonany dwoma ostatnimi metodami. Pomiary zostały wykonane dla grafów o wierzchołkach od 6 do 11 włącznie. W celu uzyskania dokładniejszych wyników każdy przypadek został wykonany 50 razy, a wynik został uśredniony.

2. Teoria.

Problem komiwojażera jest to zagadnienie optymalizacyjne polegające na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Wyróżnia się symetryczny i asymetryczny problem komiwojażera. W pierwszym dla dowolnego wierzchołka A i B koszt przejścia od pierwszego do drugiego jest taki sam jak od drugiego do pierwszego. W przypadku asymetrycznym warunek ten nie jest spełniony.

3. Wykonanie.

Program zawiera czytelne menu. Pozwala ono swobodne wykonywanie algorytmów na wczytanym grafie podstawowym, który z przyczyn losowych ma 10-wierzchołkowów. Z poziomu menu można również wyświetlić ten graf, a także uruchomić automatyczne testy.

Grafy w programie są przechowywane w macierzy incydencji. Jest ona zaimplementowana jako dwuwymiarowa tablica zawarta w abstrakcyjnej klasie AlgorithmTSP. Dane do niej są wczytywane poprzez klasę DataLoader. Pierwsza liczba w pliku tekstowym jest traktowana jako rozmiar grafu. Następne linie to zawierają dane do macierzy. Algorytm przestaje wczytywać liczbę z pliku, gdy napotka spację.

Po klasie AlgorithmTSP dziedziczą klasy obliczające najmniejszy koszt przejścia grafu, czyli BruteForce i HeldKarp. Nadpisują one czysto wirtualną metodę CalculatePath przyjmującą jako argument wierzchołek początkowy. W obu klasach są to metody inicjujące tworzące zmienne, które następnie są przekazywane do rekurencyjnej metody o tej samej nazwie.

```
void
BruteForce::CalculatePath(unsigned currentVertex, unsigned &minPrice,
unsigned currentPrice,
                          const std::vector<bool> &visitedVertexes) {
    for (unsigned vertex_index = 0; vertex_index < graphSize_;
++vertex_index) {
        if (!visitedVertexes[vertex_index]) {
            std::vector<bool> currentVisitedVertices = visitedVertexes;
            currentVisitedVertices[vertex_index] = true;
            unsigned iteration_price = currentPrice +
graph_[currentVertex][vertex_index];
            if (SetMinPriceIfLastVertex(minPrice, iteration_price,
currentVisitedVertices)) {
                return;
            }
            CalculatePath(vertex_index, minPrice, iteration_price,
currentVisitedVertices);
        }
    }
}
```

Algorytm brute force opiera się na zmiennej zawierającej index bieżącego wierzchołka startowego, a także bieżącej ceny. Ponadto do funkcji przekazywana jest referencja na vector zmiennych boolowskich, określających które wierzchołki są jeszcze dostępne do odwiedzenia, a także referencję na aktualną minimalną cenę określoną przez algorytm. Domyślnie zawiera ona maksymalną wartość dostępną dla unsigned int. W tym celu wykorzystana została biblioteka limits. Każde wywołanie sprawdza, które wierzchołki nie zostały jeszcze odwiedzone i w ich przypadku dodaje koszt od początkowego do ziterowanego i dla zaktualizowanych danych wywołuje dla niego metodę CalculatePath. Sposób ten był prosty w implementacji, jednakże wymaga optymalizacji. Dobrym pomysłem byłoby znalezienie sposobu na obejście kopiowania vectora zmiennych boolowskich w każdym wywołaniu.

```
unsigned HeldKarp::CalculatePath(unsigned startVertex, const
std::vector<bool> &visitedVertices) {
    if (CheckIfAllVerticesAreVisited(visitedVertices)) {
        return 0;
    }
    std::vector<unsigned> prices;
    for (unsigned i = 0; i < graphSize_; ++i) {
        if (!visitedVertices[i]) {
            std::vector<bool> currentVisitedVertices = visitedVertices;
            currentVisitedVertices[i] = true;
            prices.push_back(CalculatePath(i, currentVisitedVertices) +
graph_[startVertex][i]);
        }
    }
    return *std::min_element(prices.begin(), prices.end());
}
```

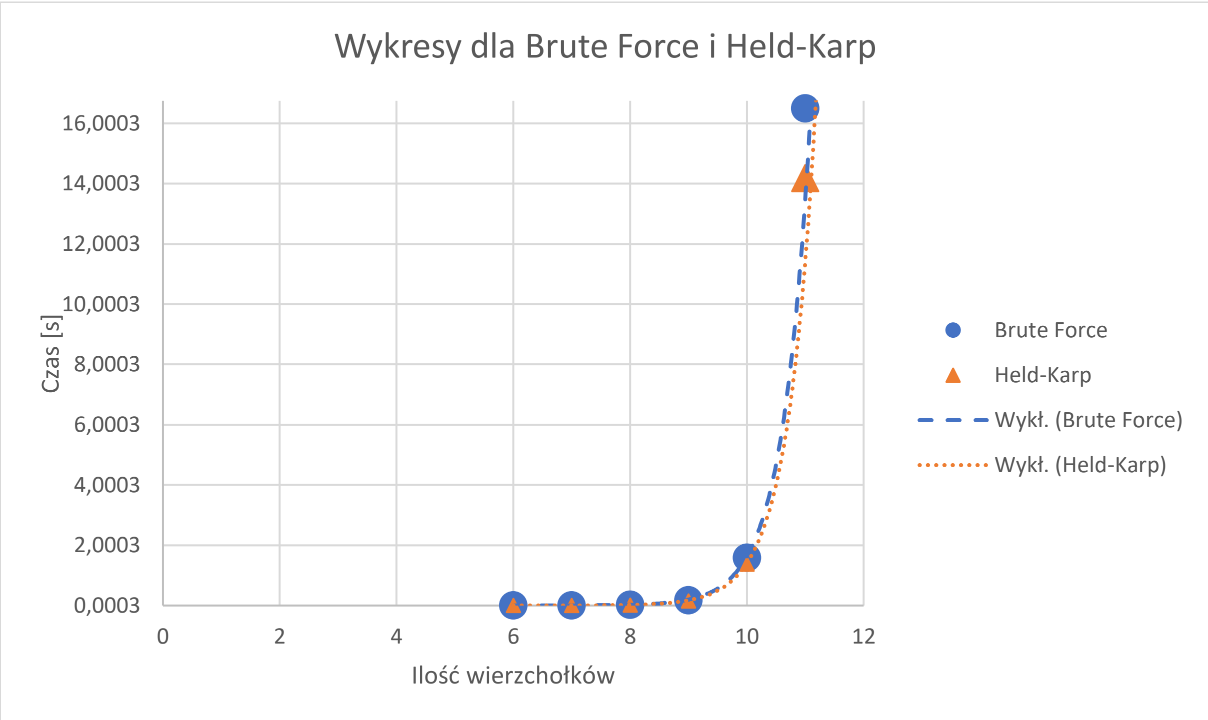
Podejście do problemu algorytmem Helda-Karpa polega na ciągłym dodawaniu dla kolejnych przypadków kosztu przejścia z wierzchołka startowego do bieżącego do kosztu zwróconego przez kolejne wywołania metody CalculatePath. W momencie, gdy wszystkie wierzchołki są odwiedzone zwracane jest 0. Dla wywołań funkcji, które nie odwiedziły wszystkich wierzchołków po przejściu po wszystkich dostępnych wierzchołkach wybierane jest najmniejszy z kosztów zwróconych i zapisanych w wektorze prices, a następnie jest zwracany. W ten sposób pierwsze wywołanie metody ma n-1 kosztów, gdzie n to liczba wierzchołków, z których wybierany jest jeden o najmniejszej wartości.

Pomiary czasu zostały wykonane za pomocą biblioteki windows.h. Jest ona wystarczająco dokładna dla rozwiązywanego problemu. Poniżej wklejone zostały metody Timer.cpp.

```
double PCFreq = 0.0;
__int64 CounterStart = 0;
```

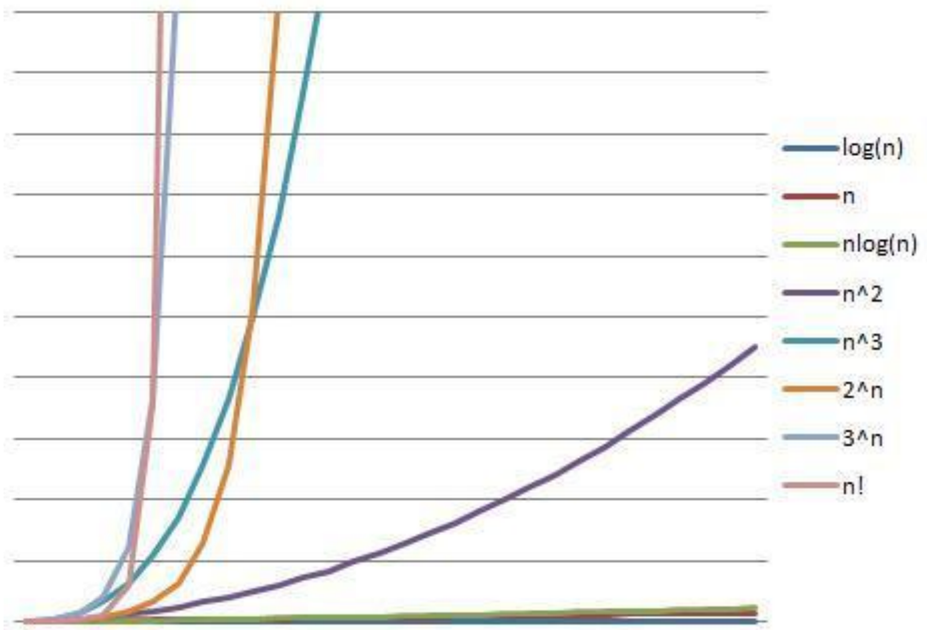
```
void Timer::StartCounter() {
    LARGE_INTEGER li;
    if (!QueryPerformanceFrequency(&li))
        std::cout << "QueryPerformanceFrequency Failed!" << std::endl;

    PCFreq = double(li.QuadPart) / 1.0;

    QueryPerformanceCounter(&li);
    CounterStart = li.QuadPart;
}

double Timer::GetCounter() {
    LARGE_INTEGER li;
    QueryPerformanceCounter(&li);
    return double(li.QuadPart - CounterStart) / PCFreq;
}
```

4. Pomiary i analiza.



Rysunek 1 Wykresy pomiarów czasu dla dwóch algorytmów.

Złożoność czasowa dla brute force jest O(n!), a dla helda-karpa $O(n^2 2^n)$. Dane przedstawione na wykresie zgadzają się zatem z założeniami teoretycznymi. Algorytm helda-karpa jest efektywniejszy czasowo. Poniżej znajduje się wykres przykładowych złożoności czasowych. Kształt uzyskanych linii trendu jest z nimi zgodny.



Rysunek 2 Poglądowe wykresy do porównania.

5. Wnioski.

Problem komiwojażera jest NP-trudny. Złożoność czasowa jego rozwiązań musi być zatem duża. W trakcie implementacji ważna była optymalizacja, choć krótki kod powodował, że ciężko było. W tym celu wektory boolowskie były przekazywane przez referencję co pozwoliło ominąć niepotrzebne kopiowanie. Dużo czasu w skali algorytmu zajmuje kopiowanie wektora wewnątrz iteracji i sprawdzanie każdorazowo czy nie aktualny wierzchołek nie jest ostatnim. W celu przyspieszenia działania kodu należałoby poszukać lepszego rozwiązania.