Projektowanie Efektywnych Algorytmów

Projekt 1

Katsiaryna Kolyshko 276708

Dr. inż. Marcin Łapuszyński

Wstęp teorytyczny

1. **Metoda brute force** (pełnego przeglądu): Algorytm brute force polega na obliczeniu kosztu dla wszystkich możliwych tras, co oznacza przeglądanie każdej permutacji miast. Dla problemu komiwojażera, liczba wszystkich możliwych tras wynosi (n−1)!(n-1)!(n−1)!, gdzie nnn to liczba miast. Ostatecznie brute force sprawdza każdą trasę i wybiera tę o najmniejszym koszcie.

Złożoność obliczeniowa: O(n!)– jest to bardzo wysoka złożoność, co czyni algorytm niepraktycznym dla większych zbiorów danych.

2. **Algorytm najbliższego sąsiada**: Algorytm ten zaczyna od dowolnego miasta, a następnie w każdej iteracji wybiera najbliższe nieodwiedzone miasto. Algorytm kończy działanie, gdy odwiedzi wszystkie miasta i wróci do początkowego.

Złożoność obliczeniowa: O(n^2)– algorytm przeszukuje macierz sąsiedztwa (macierz kosztów), co w najgorszym przypadku oznacza przeszukanie nnn wierszy i nnn kolumn, co daje kwadratową złożoność. Jest to zdecydowanie szybsze niż brute force, ale nie gwarantuje optymalnej trasy.

3. **Algorytm losowy**: W algorytmie losowym generuje się przypadkowe permutacje miast i oblicza koszt każdej z nich. Po wykonaniu ustalonej liczby iteracji wybierana jest trasa o najniższym koszcie. Skuteczność algorytmu zależy od liczby iteracji – im więcej prób, tym większa szansa na znalezienie dobrej trasy.

Złożoność obliczeniowa: O(k·n), gdzie k to liczba losowych tras (iteracji), a n to liczba miast. Algorytm działa znacznie szybciej niż brute force, ale wynik zależy od liczby prób i może nie być optymalny. W praktyce jest to kompromis między czasem obliczeń a jakością wyniku.

Podsumowanie :

- **Brute force**: O(n!)– dokładny, ale bardzo wolny.
- **Najbliższy sąsiad**: O(n^2)– szybki, ale może nie być optymalny.
- **Losowy**: O(k·n) – szybki, zależny od liczby iteracji, wynik losowy.

Z tego nieoczewiście wynika który z tych algorytmów jest najlepszy, i musimy to sprawdzić analizą wyników z naszych badań i eksperymentów.

Opis implementacji algorytmów i działąnie programu

Funkcja metody pomiaru czasu:

Aby zmierzyć i pobrać czas trwania algorytmu, użyłam funkcje QueryPerfomanceFrequency. Niżej jest przedstawiony kod działania tych funkcji:

```cpp
//Sczytywanie czasu wykonania algorytmu

double PCFreq = 0;          // Częstotliwość zegara
__int64 counterStart = 0.0; // Początek licznika

void StartCounter()
{
    LARGE_INTEGER freq;
    if (!QueryPerformanceFrequency(&freq))
        cout << "QueryPerformanceFrequency failed!\n";

    PCFreq = double(freq.QuadPart) / 1000.0;

    QueryPerformanceCounter(&freq);
    counterStart = freq.QuadPart;
}

double GetCounter()
{
    LARGE_INTEGER freq;
    QueryPerformanceCounter(&freq);
    return double(freq.QuadPart - counterStart) / PCFreq;
}
```

rysunek 1: kod funkcji StartCounter( ) i GetCounter( )

Następnie mamy funkcje calculateCost.

Funkcja calculateCost oblicza łączny koszt przejścia po zadanej trasie, reprezentowanej przez wektor miast. Przyjmuje ona jako argumenty macierz sąsiedztwa (adjMatrix), która zawiera koszty (odległości) między poszczególnymi miastami, liczbę miast (dimension) oraz wektor miast (route), który określa kolejność odwiedzanych miast. Funkcja dodaje koszty między sąsiadującymi miastami, a na końcu dodaje koszt powrotu do miasta początkowego, co zamyka cykl.

Funkcja jest kluczowa do obliczenia kosztu (długości) trasy dla każdej wygenerowanej ścieżki. Jest wykorzystywana w algorytmie Brute Force, aby ocenić koszt każdej permutacji miast, a także w algorytmach Nearest Neighbor i Random, by sprawdzić, która trasa ma najniższy koszt i jest optymalna.

```cpp
int calculateCost(vector<vector<int>> &adjMatrix, int dimension, vector<int> &route)
{

    int totalCost = 0;
    for (int i = 0; i < dimension - 1; ++i)
    {
        totalCost += adjMatrix[route[i]][route[i + 1]];
    }
    totalCost += adjMatrix[route[dimension - 1]][route[0]];
    return totalCost;
}
```

rysunek 2: kod funkcji calculateCost( )

Przejdziemy do pierwszego algorytmu, czyli *Brute Force*, albo *Pełnego Przeglądu.*

Algorytm Brute Force działa poprzez przeszukiwanie wszystkich możliwych tras pomiędzy miastami, aby znaleźć tę o najniższym koszcie. W tym celu algorytm generuje wszystkie możliwe permutacje kolejności odwiedzania miast (oprócz miasta startowego, które zawsze pozostaje na początku i końcu

trasy). W implementacji wykorzystano funkcję ‘iota’ z biblioteki standardowej C++, która wypełnia wektor miast numerami od 1 do n-1, a następnie funkcję ‘next_permutation’, która generuje kolejne permutacje miast.

Dla każdej permutacji tworzona jest pełna trasa, zaczynająca się i kończąca w mieście startowym (0), a koszt trasy jest obliczany za pomocą funkcji ‘calculateCost’. Funkcja ta sumuje odległości pomiędzy kolejnymi miastami oraz dodaje koszt powrotu do miasta początkowego. Algorytm porównuje koszt każdej trasy i zapisuje najlepszą (najtańszą) z nich.

W ten sposób algorytm przeszukuje wszystkie możliwe kombinacje tras i wybiera tę, która ma najmniejszy koszt całkowity. Niżej jest przedstawiony kod tego algorytmu:

```
//Algorytm BruteForce
void bruteForce(int dimension, vector<vector<int>> adjMatrix, vector<int> &theBestRoute, int &totalCost1, double &executionTime, string &showSteps)
{

    vector<int> cities(dimension-1);
    int minCost = INT_MAX;
    vector<int> bestRoute;
    StartCounter();

    iota(cities.begin(), cities.end(), 1);
    do
    {
        vector<int> tour = {0};
        tour.insert(tour.end(), cities.begin(), cities.end());

        int currentCost = calculateCost(adjMatrix, dimension, tour);

        if (currentCost < minCost)
        {
            minCost = currentCost;
            bestRoute = tour;
        }

        //Jezeli uzytkownik chce popatrzec na dzialanie algorytmu krok po kroku
        if (showSteps == "ON")
        {
            cout << endl
                 << "The lowest cost by now: " << minCost << endl;
            cout << "Analized route: " << endl;
            for (int city : bestRoute)
            {
                cout << city << ", ";
            }
            cout << endl;
        }
    } while (next_permutation(cities.begin(), cities.end()));
    executionTime = GetCounter();
    bestRoute.push_back(0);
    theBestRoute = bestRoute;
    totalCost1 = minCost;
}
```

rysunek 3: kod algorytmu Brute Force

Przejdziemy do algorytmu *Nearest Neighbor.*

Algorytm ‘Nearest Neighbor’ (Najbliższego Sąsiada) jest algorytmem heurystycznym, który działa w następujący sposób: zaczyna od wybranego miasta (w naszym przypadku jest to zawsze miasto o numerze 0), a następnie w każdym kroku wybiera najbliższe, jeszcze nieodwiedzone miasto. Proces ten powtarza się, aż wszystkie miasta zostaną odwiedzone. Na końcu algorytm wraca do miasta startowego, zamykając trasę.

Implementacja zaczyna się od oznaczenia miasta startowego jako odwiedzonego oraz dodania go do trasy. Następnie w pętli algorytm dla każdego kolejnego miasta szuka najbliższego sąsiada, korzystając z macierzy sąsiedztwa (adjMatrix), która przechowuje odległości pomiędzy miastami. Po znalezieniu

najbliższego miasta, dodaje je do trasy, oznacza jako odwiedzone i kontynuuje do momentu, aż wszystkie miasta zostaną odwiedzone.

Podczas działania algorytmu, jeśli użytkownik włączy opcję pokazywania kroków (showSteps), algorytm na bieżąco wyświetla aktualnie analizowaną trasę i jej koszt. Po odwiedzeniu wszystkich miast algorytm wraca do miasta startowego, zamykając cykl i oblicza całkowity koszt trasy.

Algorytm ten, choć szybki i prosty, nie zawsze znajduje rozwiązanie optymalne, ponieważ wybiera najbliższego sąsiada w każdym kroku, co może prowadzić do suboptymalnych tras.

Niżej jest przedstawiony kod algorytmu:

```
//Algorytm Heurystyczny Nearest Neighbour
void nearestNeighborTSP(const vector<vector<int>> &adjMatrix, int dimension, vector<int> &theBestRoute, int &totalCost1, double &executionTime, string &showSteps)
{
    vector<bool> visited(dimension, false);
    visited[0] = true;
    vector<int> tour;
    int totalDistance = 0;
    int currentCity = 0;
    StartCounter();
    tour.push_back(currentCity);

    for (int i = 1; i < dimension; i++)
    {
        int nearestCity = -1;
        int minDistance = numeric_limits<int>::max();

        for (int j = 0; j < dimension; j++)
        {
            if (!visited[j] && adjMatrix[currentCity][j] < minDistance)
            {
                nearestCity = j;
                minDistance = adjMatrix[currentCity][j];
            }
        }

        tour.push_back(nearestCity);
        visited[nearestCity] = true;
        totalDistance += minDistance;
        currentCity = nearestCity;

        if (showSteps == "ON")
        {
            cout << "The lowest cost by now: " << totalDistance << endl;
            cout << "Analized route: " << endl;
            for (int city : tour)
            {
                cout << city << ", ";
            }
        }
    }

    totalDistance += adjMatrix[currentCity][0];
    tour.push_back(0);

    executionTime = GetCounter();

    theBestRoute = tour;
    totalCost1 = totalDistance;
}
```

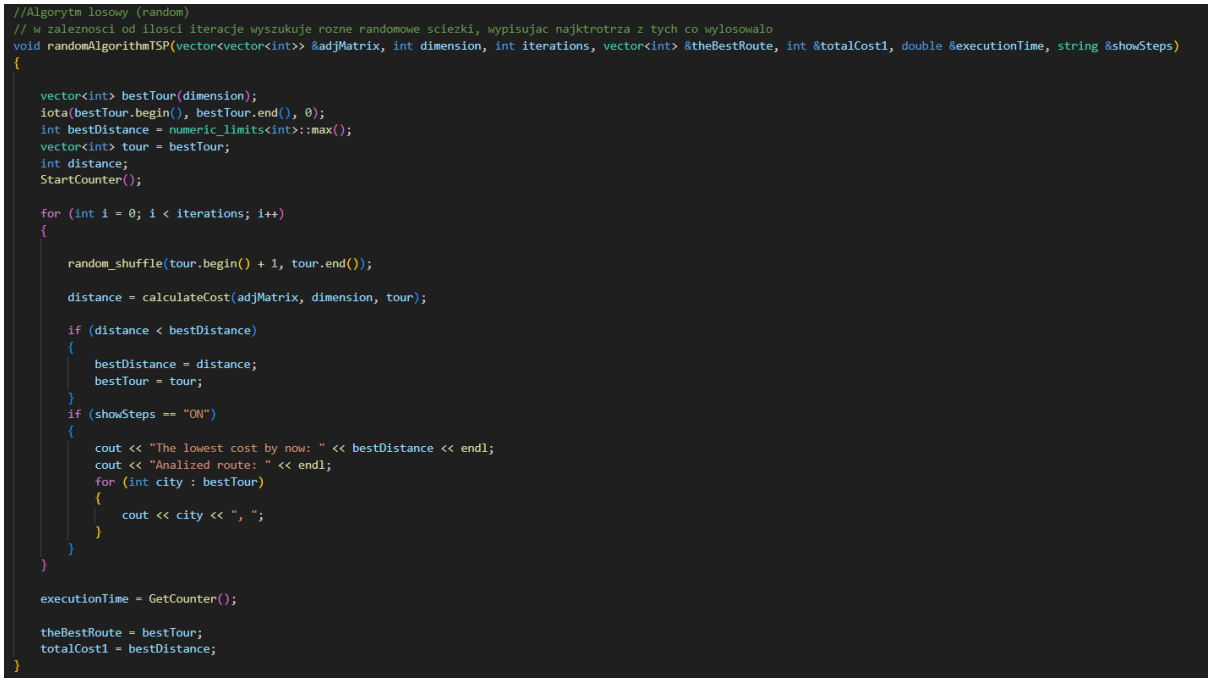
rysunek 4: kod algorytmu Nearest Neighbor

Następnie mamy algorytm *Losowy*.

Algorytm losowy (randomowy) polega na generowaniu wielu losowych tras, a następnie wybieraniu najlepszej spośród nich. Działa w ten sposób, że w każdej iteracji tworzy przypadkową permutację miast, liczy koszt tej trasy za pomocą funkcji calculateCost, a następnie porównuje go z dotychczasowym najlepszym wynikiem. Jeśli nowa trasa jest lepsza, zapisuje ją jako najlepszą.

Chociaż ten algorytm jest bardzo prosty i szybki, jego wyniki zależą od szczęścia – nie gwarantuje znalezienia optymalnej trasy, ponieważ losowe trasy mogą być dalekie od optymalnego rozwiązania.

Kod Algorytmu wygląda następnie:



```cpp
//Algorytm losowy (random)
// w zaleznosci od ilosci iteracje wyszukuje rozne randomowe sciezki, wypisujac najktrotrza z tych co wylosowalo
void randomAlgorithmTSP(vector<vector<int>> &adjMatrix, int dimension, int iterations, vector<int> &theBestRoute, int &totalCost1, double &executionTime, string &showSteps)
{

    vector<int> bestTour(dimension);
    iota(bestTour.begin(), bestTour.end(), 0);
    int bestDistance = numeric_limits<int>::max();
    vector<int> tour = bestTour;
    int distance;
    StartCounter();

    for (int i = 0; i < iterations; i++)
    {

        random_shuffle(tour.begin() + 1, tour.end());

        distance = calculateCost(adjMatrix, dimension, tour);

        if (distance < bestDistance)
        {
            bestDistance = distance;
            bestTour = tour;
        }
        if (showSteps == "ON")
        {
            cout << "The lowest cost by now: " << bestDistance << endl;
            cout << "Analized route: " << endl;
            for (int city : bestTour)
            {
                cout << city << ", ";
            }
        }
    }

    executionTime = GetCounter();

    theBestRoute = bestTour;
    totalCost1 = bestDistance;
}
```

rysunek 5: kod algorytmu Random

Funkcja configurate służy do wczytania konfiguracji algorytmu z pliku o nazwie podanej jako argument (filename). Jej zadaniem jest odczytanie ustawień, takich jak:

- INPUT_FILE: nazwa pliku z danymi wejściowymi, który zawiera macierz odległości.
- ALGORITHM: wybór algorytmu do użycia (np. BRUTE_FORCE, RANDOM, NEAREST_NEIGHBOUR).
- RANDOM_ITERATIONS: liczba iteracji dla algorytmu losowego, która jest odczytywana tylko wtedy, gdy wybrany algorytm to RANDOM.
- SHOW_STEPS: ustawienie, które określa, czy kroki algorytmu mają być wyświetlane na bieżąco.
- OUTPUT_FILE: nazwa pliku, do którego mają zostać zapisane wyniki.

Funkcja otwiera plik konfiguracyjny, czyta linie po kolei i na podstawie kluczowych słów (np. INPUT_FILE:, ALGORITHM:) przypisuje odpowiednie wartości do zmiennych. Każda z tych zmiennych jest następnie używana w programie do sterowania działaniem algorytmu oraz obsługi danych wejściowych i wyjściowych.

```cpp
void configurate(const string &filename, string &algorithm, int &randomIteractions, string &inputFile, string &showSteps, string &outputFile)
{
    ifstream file(filename);

    if (!file.is_open())
    {
        cerr << "Unable to open file: " << filename << endl;
        return;
    }

    string line;
    while (getline(file, line))
    {

        line.erase(0, line.find_first_not_of(" \t\n\r"));
        line.erase(line.find_last_not_of(" \t\n\r") + 1);

        if (line.find("INPUT_FILE:") != string::npos)
        {
            inputFile = line.substr(line.find(":") + 1);
            inputFile.erase(0, inputFile.find_first_not_of(" \t"));
        }
        if (line.find("ALGORITHM:") != string::npos)
        {
            algorithm = line.substr(line.find(":") + 1);
            algorithm.erase(0, algorithm.find_first_not_of(" \t"));
        }
        if (line.find("RANDOM_ITERATIONS:") != string::npos && algorithm == "RANDOM")

        {
            string randomIteractionsStr;
            randomIteractionsStr = line.substr(line.find(":") + 1);
            randomIteractionsStr.erase(0, randomIteractionsStr.find_first_not_of(" \t"));
            randomIteractions = stoi(randomIteractionsStr);
        }
        if (line.find("SHOW_STEPS:") != string::npos)
        {
            showSteps = line.substr(line.find(":") + 1);
            showSteps.erase(0, showSteps.find_first_not_of(" \t"));
        }
        if (line.find("OUTPUT_FILE:") != string::npos)
        {
            outputFile = line.substr(line.find(":") + 1);
            outputFile.erase(0, outputFile.find_first_not_of(" \t"));
        }
    }
    file.close();
}
```

rysunek 6: kod funkcji Configurate( )

Funkcja ‘readTSPFile’ wczytuje dane z pliku TSP. Odczytuje takie informacje jak typ problemu (TYPE), liczba miast (DIMENSION), format danych (FORMAT) i macierz odległości między miastami (DATA_SECTION). Po napotkaniu sekcji danych o wagach, wczytuje kolejne wartości, które zapisuje w wektorze ‘unresolvedGraphData’. Zakończenie odczytu następuje po napotkaniu linii "EOF". Funkcja dostarcza dane do dalszych obliczeń w algorytmach szukających optymalnej trasy.

```cpp
void readTSPFile(const string &filename, string &type, int &dimension, string &format, vector<vector<int>> &unresolvedGraphData)
{
    ifstream file(filename);

    if (!file.is_open())
    {
        cerr << "Unable to open file: " << filename << endl;
        return;
    }

    string line;
    bool inEdgeWeightSection = false;

    while (getline(file, line))
    {

        line.erase(0, line.find_first_not_of(" \t\n\r"));
        line.erase(line.find_last_not_of(" \t\n\r") + 1);

        if (line.find("TYPE:") != string::npos && type.empty())
        {
            type = line.substr(line.find(":") + 1);
            type.erase(0, type.find_first_not_of(" \t"));
        }
        else if (line.find("DIMENSION:") != string::npos && dimension == 0)
        {
            istringstream iss(line.substr(line.find(":") + 1));
            iss >> dimension;
        }
        else if (line.find("FORMAT:") != string::npos && format.empty())
        {
            format = line.substr(line.find(":") + 1);
            format.erase(0, format.find_first_not_of(" \t"));
        }
        else if (line.find("DATA_SECTION") != string::npos)
        {
            inEdgeWeightSection = true;
            continue;
        }

        if (inEdgeWeightSection)
        {
            if (line.find("EOF") != string::npos)
            {
                break;
            }

            vector<int> weightsRow;
            istringstream iss(line);
            int weight;

            while (iss >> weight)
            {
                weightsRow.push_back(weight);
            }

            unresolvedGraphData.push_back(weightsRow);
        }
    }

    file.close();
}
```

rysunek 7: kod funkcji readTSPFile( )

Funkcja ‘saveResults’ zapisuje wyniki działania algorytmu do pliku CSV. Przyjmuje nazwę pliku, typ problemu, nazwę użytego algorytmu, najlepszą trasę, minimalny koszt (odległość) oraz czas wykonania. Otwiera plik o podanej nazwie, a następnie zapisuje wiersz z informacjami: typ, algorytm, trasa, minimalny koszt oraz czas wykonania w milisekundach. Na koniec zamyka plik. Jeśli pliku nie da się otworzyć, funkcja wyświetla komunikat o błędzie.

Kod wygląda następnym sposobem:

```cpp
void saveResults(string outputFileName, string type, string algorithm, vector<int> bestRoute, int minCost, double time)
{
    ofstream outputFile(outputFileName);
    if (outputFile.is_open())
    {
        outputFile << "Type, Algorithm Used, Best Tour, Minimum Distance, Time Of Execution\n";
        outputFile << type << ", ";
        outputFile << algorithm << ", ";
        for (int city : bestRoute)
        {
            outputFile << city << " ";
        }
        outputFile << ", " << minCost;
        outputFile << ", " << time << "ms"
                                      "\n";
        outputFile.close();
        cout << "Results written to tsp_results.csv" << endl;
    }
    else
    {
        cerr << "Unable to open file for writing." << endl;
    }
}
```

rysunek 8: kod funkcji saveResults( )

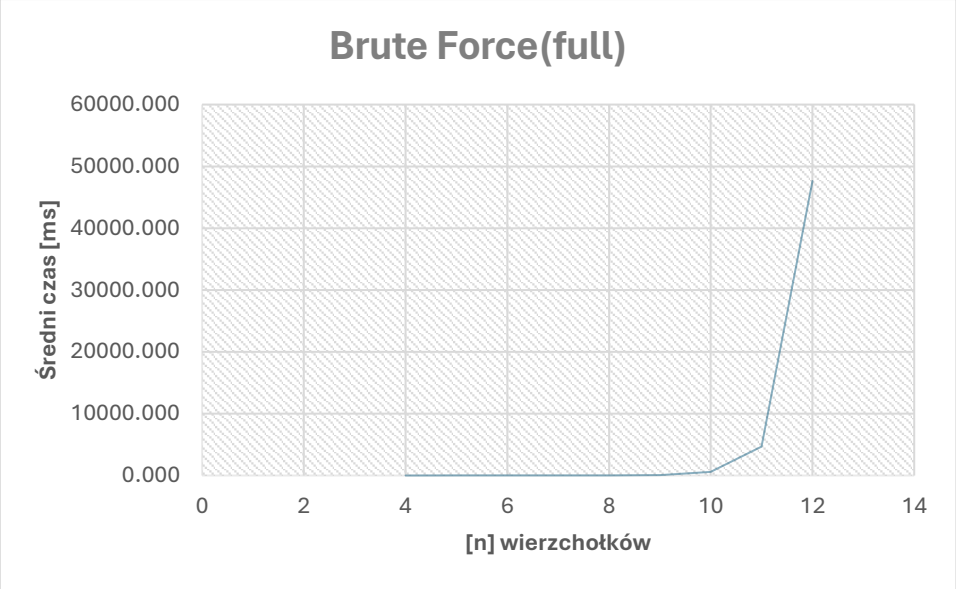
Analiza wyników:

Niżej są przedstawione wyniki pomiarów czasu dla różnych algorytmów:

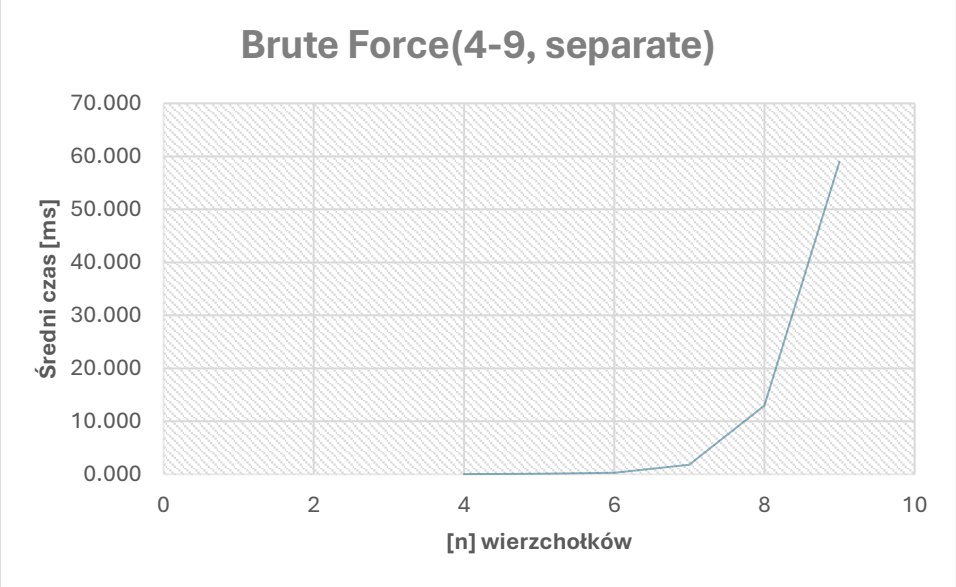
Algorytm ‘Brute Force’:

Brute Force	
n wierzchołków	Średni czas [ms]
4	0.038
5	0.127
6	0.295
7	1.772
8	12.990
9	59.004
10	568.541
11	4691.114
12	47638.234

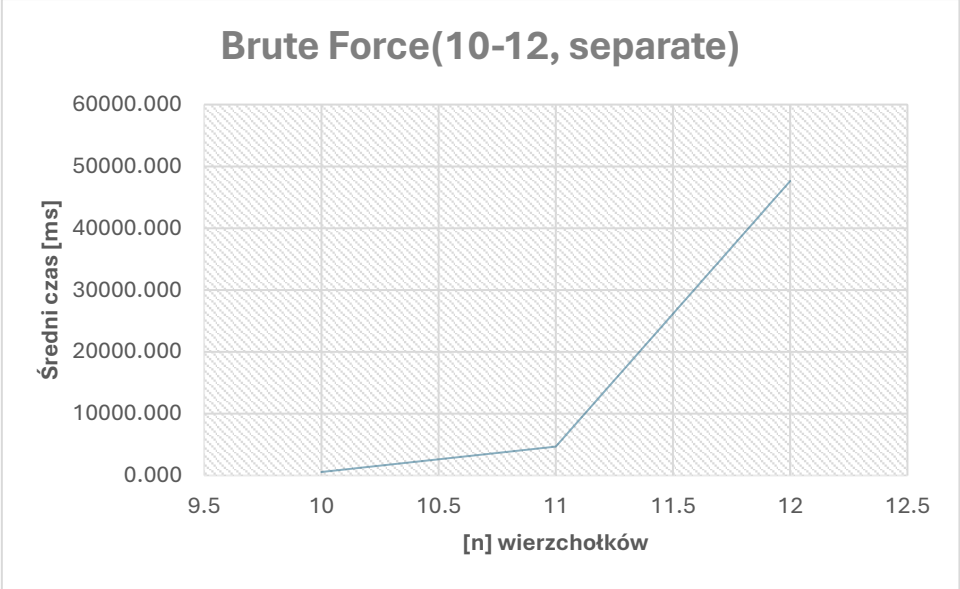
tablica 1: średni czas wykonania algorytmu Brute Force



Grafik 1: Pełny grafik dla algorytmu Brute Force



Grafik 2: Grafik dla algorytmu Brute Force, liczba wierz. 4-9



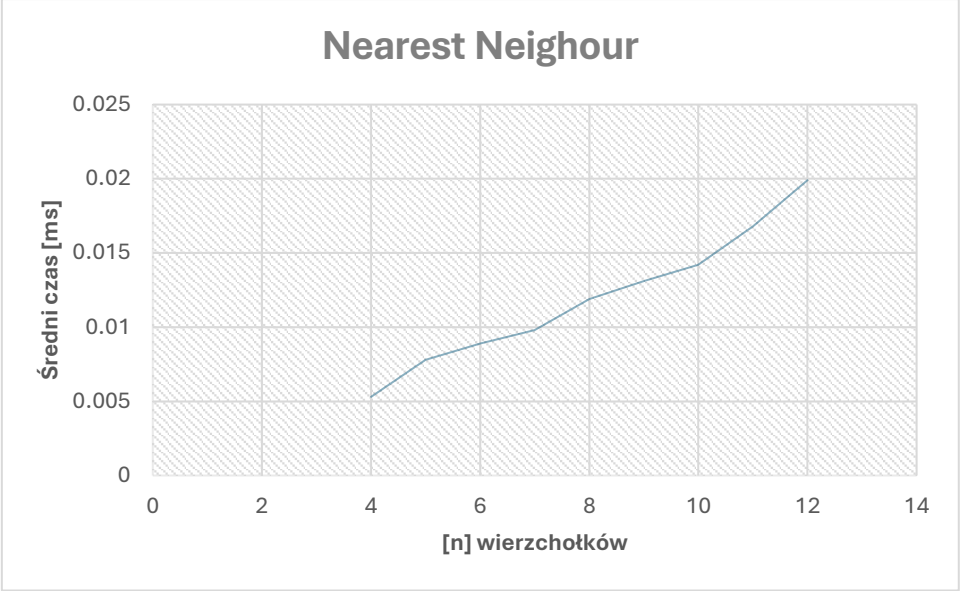
Grafik 3: Grafik dla algorytmu Brute Force, liczba wierz. 10-12

Możemy od razu zauważyć, że czas bardzo szybko się wzrasta, co i było oczekiwano od algorytmu Brute Force.

Przejdziemy do algorytmu ‘Nearest Neighbour’:

Nearest Neighbour	
n wierzchołków	Średni czas [ms]
4	0.0053
5	0.0078
6	0.0089
7	0.0098
8	0.0119
9	0.0131
10	0.0142
11	0.0168
12	0.0199

tablica 2: średni czas wykonania algorytmu Nearest Neightbour

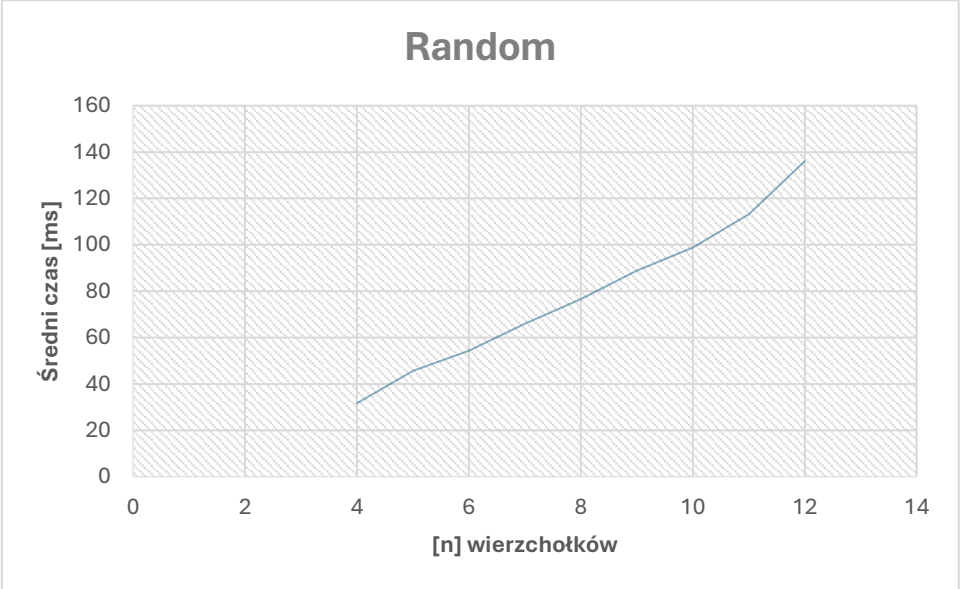


Grafik 4: Grafik dla algorytmu Nearest Neghbour

Wyniki algorytmu Random (100000 iteracji):

Random	
n wierzchołków	Średni czas [ms]
4	31.697
5	45.589
6	54.288
7	65.896
8	76.627
9	88.911
10	98.782
11	113.153
12	136.047

tablica 3: średni czas wykonania algorytmu Random



Grafik 5: Grafik dla algorytmu Random

Metoda Brute Force:

Złożoność czasowa (O(n!)) dla tego algorytmu jest potwierdzona wynikami. Wraz ze wzrostem liczby miast (n), czas obliczeń rośnie wykładniczo, co jest zgodne z przewidywaniami teoretycznymi. Graficzne przedstawienie wyników wyraźnie pokazuje bardzo szybki wzrost czasu, a algorytm staje się niepraktyczny dla więcej niż 10 miast ze względu na zbyt długie czasy wykonywania (np. ponad 47 sekund dla 12 miast). Jest to zgodne z oczekiwanym zachowaniem dla metody brute force.

Algorytm Najbliższego Sąsiada:

Złożoność czasowa ($O(n^2)$) dla algorytmu Najbliższego Sąsiada również znajduje potwierdzenie w wynikach. Czas wykonywania rośnie stopniowo wraz ze wzrostem liczby miast, ale pozostaje na akceptowalnym poziomie nawet przy większych zbiorach danych, co pokazują wyniki, gdzie czas jest minimalny (np. 0.0199 ms dla 12 miast). Jednak, zgodnie z teorią, mimo że jest szybszy, nie gwarantuje znalezienia trasy optymalnej, i konieczne są dalsze badania nad kosztami wynikowymi.

Algorytm Losowy:

Dla algorytmu Losowego, złożoność \(O(k \cdot n)\) zależy od liczby iteracji ((k)). Przy 100,000 iteracjach czas działania algorytmu wzrasta liniowo wraz z liczbą miast, co jest zgodne z oczekiwaniami. Wyniki graficzne pokazują stopniowy wzrost, a mimo że działa szybciej niż brute force, jego wydajność nadal jest niższa niż w przypadku Najbliższego Sąsiada. Ze względu na losowość, algorytm czasami znajduje rozwiązania bliskie optymalnym, ale jest to silnie zależne od liczby prób, co jest zgodne z teorią.

Wyniki:

Wyniki eksperymentalne są zgodne z przewidywaniami teoretycznymi. Metoda brute force zachowuje się zgodnie z oczekiwaniami, z wykładniczym wzrostem czasu, Najbliższy Sąsiad oferuje szybsze, ale nieoptymalne rozwiązanie, a algorytm Losowy, choć zmienny, wykazuje liniową złożoność czasową, jak przewidywano. Projekt skutecznie pokazuje kompromis pomiędzy dokładnością a wydajnością w różnych algorytmach.

Wnioski:

Chciałabym dodać trochę informacje o tym jak program sczytuje informacje pliku. Najpierw musimy zmienić informacje w pliku tekstowym „config.txt”, a po za tym wczytuje następną informacje: nazwa pliku z którego będziemy pobierać matryce, algorytm który chcemy sprawdzić, liczba iteracji (jeżeli wybrany algorytm jest „Random”), wczytuje czy musi pokazywać kroki działania algorytmu, nazwę pliku w którym będą zapisane wyniki.

Następnie przechodzimy do pliku, z którego wczytujemy macierz. Z tego dokumenty „entryData.txt” pobieramy następne informacje: typ problemu (TSP, czyli travelling salesman problem; jeżeli typ jest atsp – pokazuje błąd), ilość miast, format informacji (może być matryca, może być lista), macierz i ostatecznie „EOF” oznacza, że to jest koniec pliku, czyli „End Of File”.

Program wczytuje informacje jako macierz, ale też może sczytywać informacje odległości Euklidesowej. Różnica między wczytywaniem danych z macierzy a obliczaniem odległości Euklidesowej polega na sposobie reprezentacji odległości między miastami. W przypadku macierzy wag odległości między miastami są już podane i zapisane w formie macierzy, gdzie każdy element

wskazuje odległość między danymi miastami. Natomiast w podejściu euklidesowym odległości są obliczane dynamicznie na podstawie współrzędnych miast za pomocą wzoru na odległość euklidesową. Macierz wag jest gotowa do użycia, a w przypadku Euklidesa konieczne jest dodatkowe obliczenie dystansów.

Oprócz tego, permutacje obejmują wszystkie możliwe sposoby podróżowania między miastami, a dla każdej permutacji funkcja ‘calculateCost; użyje dokładnych wartości z adjMatrix[i][j], uwzględniając różnice kierunkowe w odległościach, czyli opracuje macierzy symetryczne i asymetryczne.