Projektowanie Efektywnych Algorytmów

Algorytm Brute Force
oraz
programowanie dynamiczne - Algorytm Bellmana-Helda-Karpa

Kasper Radom 264023

24 XI 2023

Spis treści

1 Sformułowanie zadania

Zadane polega na opracowaniu, implementacji i analizie efektywności algorytmu przeglądu zupełnego oraz programowania dynamicznego, na przykładzie asymetrycznego problem komiwojażera.

Problem komiwojażera polega na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym.

Cykl Hamiltona jest ścieżką przechodząca dokładnie jeden raz przez każdy wierzchołek i wracająca do wierzchołka początkowego.

2 Metoda

2.1 Przegląd zupełny

Metoda przeglądu zupełnego, znana również jako przeszukiwanie wyczerpujące lub metoda siłowa, stanowi jedno z podejść do rozwiązania problemu komiwojażera. W ramach tego podejścia dokładnie analizuje się wszystkie możliwe ścieżki spełniające kryteria problemu. Proces ten obejmuje kompleksową eksplorację wszystkich dostępnych rozwiązań, a mianowicie odnalezienie i sprawdzenie każdego dopuszczalnego rozwiązania, obliczenie wartości funkcji celu dla każdego z nich, a następnie dokonanie wyboru ścieżki, która charakteryzuje się najmniejszą sumaryczną długością. Suma ta odpowiada sumie odległości między odwiedzonymi miejscami, czyli sumie krawędzi między poszczególnymi wierzchołkami grafu, tworzącymi trasę komiwojażera.

Główną zaletą metody przeglądu zupełnego jest jej prostota i deterministyczność. Algorytm oparty na tej metodzie jest stosunkowo łatwy do implementacji i nie wymaga zaawansowanych technik optymalizacji. Jednakże istnieje istotna wada związana z tą metodą - jest to jej ogromna złożoność obliczeniowa. Ponieważ analizuje się wszystkie możliwe kombinacje tras, czas potrzebny na rozwiązanie problemu komiwojażera za pomocą metody przeglądu zupełnego znacząco rośnie wraz z liczbą odwiedzanych miejsc (złożoność czasowa wynosi **O(n!)**). Dlatego metoda ta jest praktyczna tylko w przypadku małych zbiorów danych.

Główną zaletą metody przeglądu zupełnego jest jej prostota i deterministyczność. Algorytm oparty na tej metodzie jest stosunkowo łatwy do implementacji i nie wymaga zaawansowanych technik optymalizacji. Jednakże

istnieje istotna wada związana z tą metodą - jest to jej ogromna złożoność obliczeniowa. Ponieważ analizuje się wszystkie możliwe kombinacje tras, czas potrzebny na rozwiązanie problemu komiwojażera za pomocą metody przeglądu zupełnego znacząco rośnie wraz z liczbą odwiedzanych miejsc (złożoność czasowa wynosi **O(n!)**). Dlatego metoda ta jest praktyczna tylko w przypadku małych zbiorów danych.

W rezultacie, choć metoda przeglądu zupełnego jest zawsze skuteczna w rozwiązywaniu problemu komiwojażera, jej zastosowanie jest ograniczone i nie jest praktyczne dla dużych instancji problemu. Do rozwiązywania większych problemów komiwojażera używa się bardziej zaawansowanych i efektywnych algorytmów, które pozwalają znaleźć dokładne lub zbliżone rozwiązania w znacznie krótszym czasie.

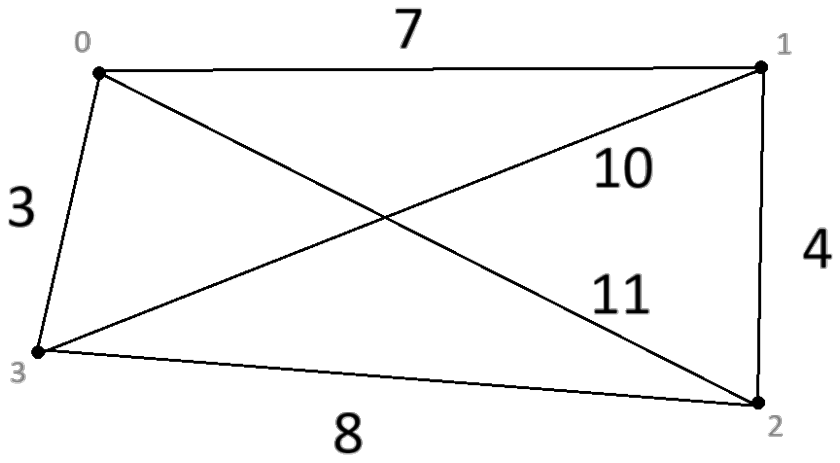
2.2 Programowanie dynamiczne

Programowanie dynamiczne polega na dzieleniu problemu na podproblemy, rozwiązywaniu ich oraz zapamiętywaniu wyników.

Algorytm Bellmana-Helda-Karpa znany też jako Algorytm Helda-Karpa opiera swoją zasadę działania na programowaniu dynamicznym. Polega on na obliczaniu kolejno długości ścieżek składających się z coraz większej liczby wierzchołków i wybieraniu tej optymalnej.

2.2.1 Przykład działania

Sposób działania algorytmu zademonstrowany jest na przykładzie 4-wierzchołkowej instancji:



Przez $d_{i,j}$ oznaczać będziemy długość krawędzi z wierzchołka i do wierzchołka j. Natomiast *D(S,x)*, gdzie x należy do zbioru S, oznacza optymalną ścieżkę od wierzchołka *0* do wierzchołka x przechodzącą przez wszystkie wierzchołki ze zbioru S.

$$d_{0,1} = d_{1,0} = 7,$$
$$d_{0,2} = d_{2,0} = 11,$$
$$d_{0,3} = d_{3,0} = 3,$$
$$d_{1,2} = d_{2,1} = 4,$$
$$d_{1,3} = d_{3,1} = 10,$$
$$d_{2,3} = d_{3,2} = 8$$

Dla $|S| = 1$:

$$D(\{1\}, 1) = d_{0,1} = d_{1,0} = 7,$$
$$D(\{2\}, 2) = d_{0,2} = d_{2,0} = 11,$$
$$D(\{3\}, 3) = d_{0,3} = d_{3,0} = 3,$$

Kolejna iteracja pozwoli nam obliczyć koszt wszystkich ścieżek 3-wierzchołkowych z wierzchołka *0*.
Dla $|S| = 2$:

$$D(\{1,2\}, 1) = D(\{2\}, 2) + d_{2,1} = 11 + 4 = 15,$$
$$D(\{1,2\}, 2) = D(\{1\}, 1) + d_{1,2} = 7 + 4 = 11,$$
$$D(\{1,3\}, 1) = D(\{3\}, 3) + d_{3,1} = 3 + 10 = 13,$$
$$D(\{1,3\}, 3) = D(\{1\}, 1) + d_{1,3} = 7 + 10 = 17,$$
$$D(\{2,3\}, 2) = D(\{3\}, 3) + d_{3,2} = 3 + 8 = 11,$$
$$D(\{2,3\}, 3) = D(\{2\}, 2) + d_{2,3} = 11 + 8 = 19$$

Przy tej iteracji odbywa się pierwsze uproszczenie względem przeglądu zupełnego. Od tego momentu rozpatrywane będą wyłącznie te 4-elementowe ścieżki, których nie można zastąpić lepszymi. Dla $|S| = 3$:

$$D(\{1,2,3\},1) = min(D(\{2,3\},2)+d_{2,1}, D(\{2,3\},3)+d_{3,1}) = min(11+4, 19+10)$$
$$= min(15, 29) = 15,$$
$$D(\{1,2,3\},2) = min(D(\{1,3\},1)+d_{1,2}, D(\{1,3\},3)+d_{3,2}) = min(13+4, 17+8)$$
$$= min(17, 25) = 17,$$
$$D(\{1,2,3\},3) = min(D(\{1,2\},1)+d_{1,3}, D(\{1,2\},2)+d_{2,3}) = min(15+10, 11+8)$$
$$= min(25, 19) = 19$$

Teraz jesteśmy w stanie już wyznaczyć rozwiązanie całego problemu dodając do otrzymanych wyników odpowiednie krawędzie prowadzące do wierzchołka początkowego i znajdując najmniejszy wynik:

$$min(D(\{1,2,3\},1) + d_{1,0}, D(\{1,2,3\},2) + d_{2,0}, D(\{1,2,3\},3) + d_{3,0})$$
$$= min(15 + 7, 17 + 11, 19 + 3) = min(22, 28, 22) = 22$$

Graf podany w przykładzie jest symetryczny, więc zgodnie z przewidywaniami algorytm pozwala znaleźć dwa optymalne rozwiązania. *0-3-2-1-0* oraz *0-1-2-3-0* o długości 22.

3 Opis implementacji algorytmu

3.1 Przegląd zupełny

Przegląd zupełny uruchamiany jest metodą *start*. Wtedy również zaczyna być mierzony czas. Owa funkcja tworzy wskaźnik na tablicę oraz przypisuje jej kolejne indeksu wierzchołków grafu. Następnie uruchamiane jest generowanie permutacji oraz szukanie optymalnego rozwiązania. Po ustawieniu ostatniego elementu tablicy przechowującej najlepszą ścieżkę na zero, pamięć zajmowana przez tablicę zostaje zwolniona.

```
void BruteForce::start() {
    int *arr = new int[matrixSize];

    for (int i = 0; i < matrixSize; i++) {
        arr[i] = i;
    }

    perm( numbers: arr);
    minPath[matrixSize] = minPath[0];
    delete[] arr;
}
```

Algorytm został zaimplementowany poprzez przeszukiwanie kolejnych permutacji tablicy *numbers* zawierającej kolejne indeksy wierzchołków grafu. Permutacje są generowane poprzez odpowiednie, rekurencyjne zamienianie miejscami elementów tablicy. W celu przyspieszenia działania algorytmu i pominięcia zbędnych kroków, permutacje zaczynające się od indeksu innego niż zero są pomijane. Ustalanie konkretnego wierzchołka jako początek ścieżki jest poprawne ponieważ cykl w grafie musi zaczynać się i kończyć tym samym wierzchołkiem, więc dowolny z nich może zostać zapisany jako początkowy.

```
void BruteForce::perm(int *numbers) {
    perm(numbers, index: 0);
}

void BruteForce::perm(int *numbers, int index) {
    int n = matrixSize;
    //
    if (index == n) {
        checkPath( arr: numbers);
        return;
    }

    for (int i = index; i < n; i++) {
        if(numbers[0] >0) return;
        swap(numbers, i: index, j: i);
        perm(numbers, index: index + 1);
        swap(numbers, i: index, j: i);
    }
}

void BruteForce::swap(int *numbers, int i, int j) {
    int temp = numbers[i];
    numbers[i] = numbers[j];
    numbers[j] = temp;
}
```

Po każdym wygenerowaniu permutacji, metoda *checkPath* sprawdza, czy ścieżka istnieje, jednocześnie obliczając jej długość. W każdej iteracji, gdy dodawany jest kolejny wierzchołek, funkcja sprawdza, czy aktualna długość przekroczyła obecnie najlepszy wynik. Jeśli tak, funkcja przechodzi do kolejnej permutacji. Na końcu sprawdzane jest, czy obecna ścieżka jest krótsza niż poprzednio zapisana. Jeśli jest krótsza niż *pathLength*, nowa długość zostaje zapisana jako *pathLength*, a nowa tablica wierzchołków jest przechowywana w zmiennej *minPath*.

```
void BruteForce::checkPath(int *arr) {
    int pathLenght = 0;
    int currentCity;
    int nextCity;

    for (int index = 0; index < matrixSize - 1; index++) {
        currentCity = arr[index];
        nextCity = arr[index + 1];

        pathLenght += matrix[currentCity][nextCity];
        if(pathLenght>=minLenght) return;
    }

    currentCity = arr[matrixSize - 1];
    nextCity = arr[0];
    pathLenght += matrix[currentCity][nextCity];

    if (pathLenght < minLenght) {
        minLenght = pathLenght;

        for(int i=0; i<matrixSize; i++){
            minPath[i] = arr[i];
        }
    }
}
```

3.2 Programowanie dynamiczne

Algorytm Helda-Karpa również jest uruchamiany przez wywołanie metody *start*. Funkcja ta na początku tworzy C w której każdej parze (*std::pair¡¿*) liczb typu *int* przyporządkowana jest inna para tego samego typu. C ma postać *(S, c):(v,p)*. Liczba S jest binarną reprezentacją zbioru indeksów wybranych wierzchołków grafu. Na przykład jeśli

$$S = 12_10 = 1100_2$$

oznacza to, że zbiór S zawiera liczby 2 oraz 3 ponieważ na drugim i trzecim miejscu binarnej reprezentacji liczby znajdują się jedynki. c jest numerem końcowego wierzchołka ścieżki stworzonej z elementów zbioru S. Jest to analogią do funkcji D ze wzorów teoretycznych przedstawionych w punkcie 2.2.1. v oznacza wagę ścieżki zdefiniowanej wcześniej przez zbiór S oraz zmienną c. Ostatnim parametrem zawartym w zmiennej C jest p. Jest to numer indeksu przedostatniego wierzchołka ścieżki opisywanej przez parę zawartą w mapie.

Mapa jest na początku uzupełniana przez krawędzie wychodzące z wierzchołka początkowego. Następnie dla każdego rozmiaru zbioru S generowane są kombinacje wierzchołków a nastęonie wyliczane parametry zapisywane są do zmiennej C.

```
void HeldKarp::start() {
    map<pair<int, int>, pair<int, int>> C;
    for (int k = 1; k < matrixSize; k++) {
        C[make_pair( x: 1 << k,  &: k)] = make_pair( &: matrix[0][k],  y: 0);
    }
    vector<int> input;
    for (int i = 1; i < matrixSize; i++) {
        input.push_back(i);
    }
    for (int subsetSize = 2; subsetSize < matrixSize; subsetSize++) {
        for (const vector<int> subset: generateCombinations(input,  k: subsetSize)) {
            int bits = 0;
            for (int bit: subset) {
                bits |= (1 << bit);
            }
            for (int k: subset) {
                int prev = bits & ~(1 << k);
                vector<pair<int, int>> res;
                for (int m: subset) {
                    if (m == 0 or m == k)continue;
                    res.push_back(make_pair(
                            x: C[make_pair( &: prev,  &: m)].first + matrix[m][k],  &: m));
                }
                int minIndex = 0;
                int min = res[minIndex].first;
                for (int i = 1; i < res.size(); ++i) {
                    if (res[i].first < min) {
                        minIndex = i;
                        min = res[i].first;
                    }
                }
                C[make_pair( &: bits,  &: k)] = res[minIndex];
            }
        }
    }
```

Po uzupełnieniu mapy C wymaganymi parametrami funkcja zaczyna wyliczać rozwiązanie iterując po węzłach w tył aż do węzła początkowego. Ścieżka oraz długość cyklu zapisywane są w zmiennych globalnych i stamtąd pobierane przez użytkownika.

```
    int bits = pow( x: 2,  y: matrixSize) - 2;
    vector<pair<int, int>> res;
    for (int k = 1; k < matrixSize; k++) {
        res.push_back(make_pair(
                x: C[make_pair( &: bits,  &: k)].first
                    + matrix[k][0],  &: k));
    }
    int opt = res[0].first;
    int parent = res[0].second;
    for (int i = 1; i < res.size(); i++) {
        if (res[i].first < opt) {
            opt = res[i].first;
            parent = res[i].second;
        }
    }
    vector<int> path;
    int new_bits;
    for (int i = 0; i < matrixSize - 1; i++) {
        path.push_back(parent);
        new_bits = bits & ~(1 << parent);
        parent = C[make_pair( &: bits,  &: parent)].second;
        bits = new_bits;
    }
    path.push_back(0);
    minLenght = opt;
    for (int i = matrixSize - 1; i >= 0; i--) {
        minPath[matrixSize - i - 1] = path[i];
    }
    minPath[matrixSize] = 0;
}
```

Funkcja *start* wywołuje metodę *generateCombinations*, która rekurencyjnie wylicza kolejne kombinacje liczb z wektora *input* o wielkości *k*.

```
vector<vector<int>> HeldKarp::generateCombinations(const vector<int> input, int k) {
    vector<int> combination( n: k,  value: 0);
    return generateCombinations(input, k,  &: combination,  index: 0,  i: 0);
}

vector<vector<int>> HeldKarp::generateCombinations
(const vector<int> &input, int k, vector<int> &combination, int index, int i) {
    vector<vector<int>> result;
    if (index == k) {
        result.push_back(combination);
        return result;
    }
    while (i < input.size() && index < k) {
        combination[index] = input[i];
        vector<vector<int>> subCombinations =
                generateCombinations(input, k,  &: combination,  index: index + 1,  i: i + 1);
        result.insert( position: result.end(),
                       first: subCombinations.begin(),  last: subCombinations.end());
        ++i;
    }
    return result;
}
```

4 Metoda testowania

Testowanie zostało zaimlementowane w metodzie *startAutoTesting*. Pobiera ona od użytkownika maksymalną liczbę wierzchołków jaką ma zawierać graf (*maxLen*) oraz ilość instancji generowanych dla każdej liczby wierzchołków (*instances*). Następnie dla każdej liczby wierzchołków generowana jest podana liczba wierzchołków poprzez wywołanie metody *generateData* losującej długość krawędzi między dwoma wierzchołkami z zakresu od wartości 1 do podanej jako argument liczby. dla każdego grafu uruchamiany jest algorytm *start* i mierzony czas wykonania tej metody.

```
void Test::startAutoTesting(){
    int maxLen;
    int instances;
    ofstream excelFile( s: "dane.csv");
    if (!excelFile.is_open()) {
        cout << "Nie można otworzyć pliku Excela." << endl;
        return;
    }
    cout << "Testy będą się zaczynały od grafu o 2 wierzchołkach"
            " do grafu o rozmiarze podanym przez Ciebie." << endl
    << "Jaki jest maksymalny rozmiar?"<< endl;
    cin >> maxLen;
    cout<<endl;
    cout << "Po ile grafów ma być wygenerowanych dla każdego kolejnego rozmiaru?"<<endl;
    cin >> instances;
    cout<<endl;
    for(int len=2; len<=maxLen;len++){
        cout << len<<endl;
        matrixSize = len;
        excelFile << len << ";;";
        for(int i=0; i<instances;i++){
            cout<< "\t"<<i<<endl;
            generateData( maxLen: 100);
            BruteForce* bruteForce = new BruteForce(matrix, matrixSize);
            time.start();
            bruteForce->start();
            time.stop();
            excelFile << time.getTime() << ";";
        }
        excelFile<<endl;
    }
    excelFile.close();
}
```

```
void Test::generateData(int maxLen) {
    matrix = new int *[matrixSize];
    for (int i = 0; i < matrixSize; ++i) {
        matrix[i] = new int[matrixSize];
    }
    for (int i = 0; i < matrixSize; i++) {
        for (int j = 0; j < matrixSize; j++) {
            if (i == j) {
                matrix[i][j] = -1;
                continue;
            }
            matrix[i][j] = rand() % maxLen + 1;
        }
    }
}
```

Mierzenie czasu zostało wdrożone przez stworzenie klasy zawierającej metody *start*, zapisującej do zmiennej *startTime* obecny czas w formie obiektu reprezentującego czas systemowy, *stop*, zapisującej kolejny raz czas oraz *getTime* zwracający różnice tych wartości w formie unsigned long long. Tak duża dokładność została wybrana dla jak najlepszego reprezentowania czasu dla małych grafów zawierających kilka wierzchołków na potrzeby stworzenia sprawozdania.

```
#include "Time.h"
void Time::start() {
    startTime = high_resolution_clock::now();
}
void Time::stop() {
    stopTime = high_resolution_clock::now();
}
unsigned long long Time::getTime() {
    return duration_cast<nanoseconds>( d: stopTime - startTime).count();
}
```

Zebrane dane zapisywane są do pliku o csv. Jeśli plik nie istnieje jest stwarzany w folderze w którym znajduje się plik wykonalny. Testowanie al-

gorytmu Helda-Karpa działa analogicznie.

5 Wyniki eksperymentów

Tabela reprezentuje eksperymenty dla poszczególnej liczby wierzchołków w grafie oraz średni czas wykonania algorytmu. Eksperyment został przeprowadzony na stu instancjach dla każdej liczby wierzchołków.

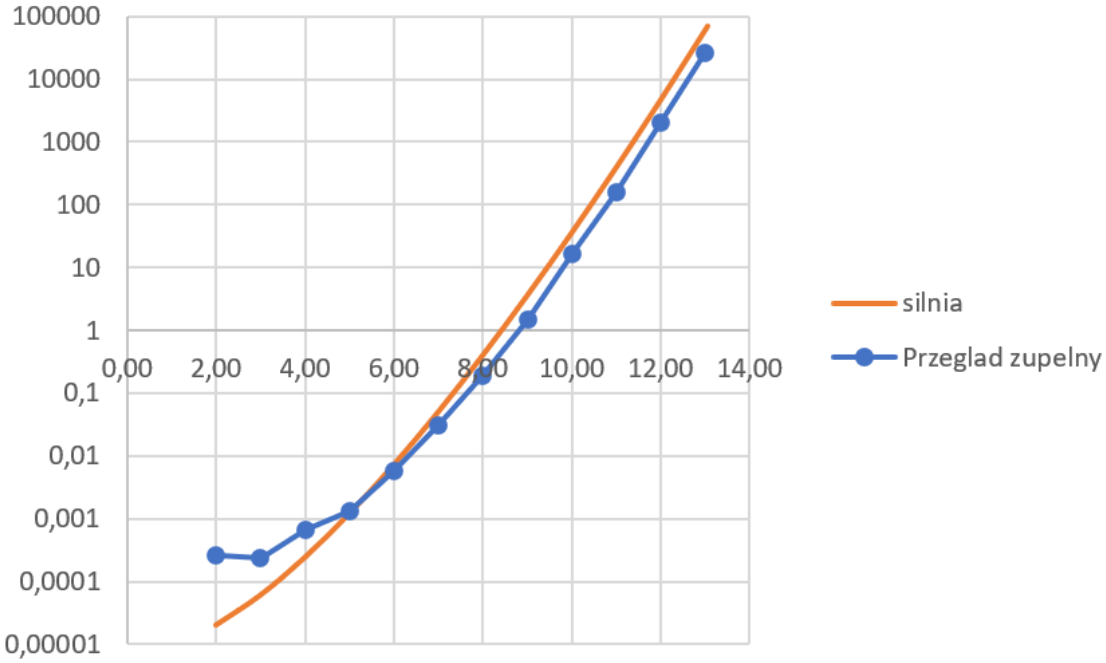
Przegląd zupełny:

Liczba wierzchołków	Uśredniony czas wykonania algorytmu [ms]
2	0,00026
3	0,00024
4	0,00065
5	0,00133
6	0,00568
7	0,03002
8	0,19294
9	1,49529
10	16,51434
11	157,03228
12	1998,06939
13	25500,78929

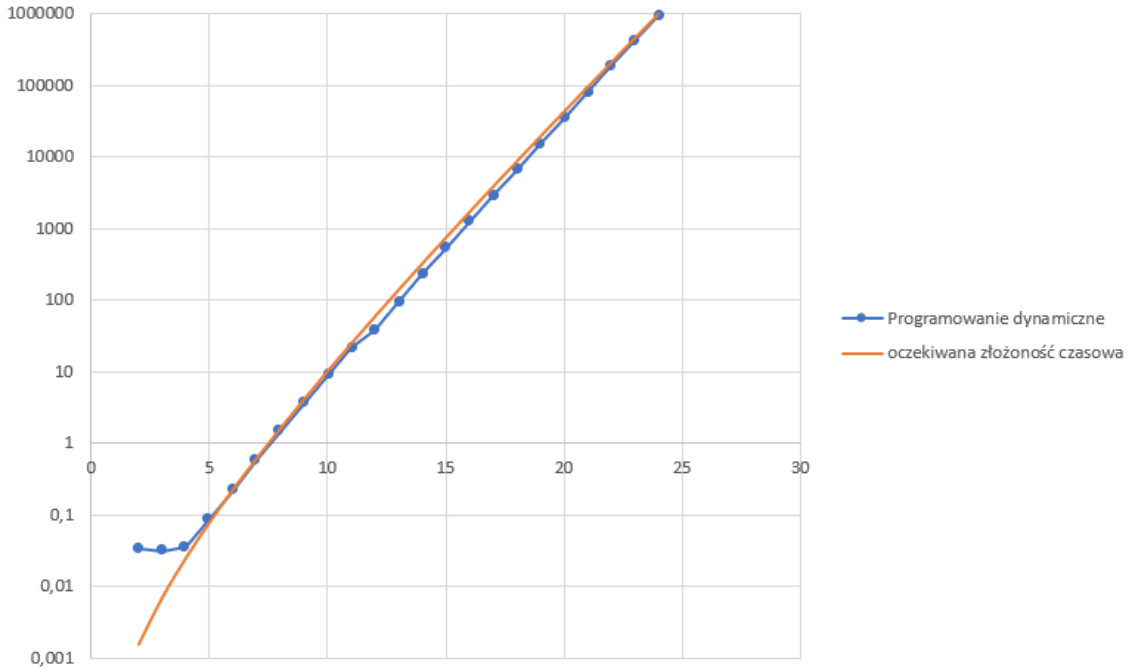
Dynamiczne programowanie:

Liczba wierzchołków	Uśredniony czas wykonania algorytmu [ms]
2	0,0339
3	0,0315
4	0,0356
5	0,0869
6	0,2183
7	0,5867
8	1,4576
9	3,6261
10	8,926
11	21,5292
12	37,7841
13	92,4514
14	226,5653
15	528,1099
16	1245,2658
17	2851,7174
18	6532,324
19	14772,027
20	34420,1847
21	77934,9396
22	184918,5695
23	411068,2206
24	929773,662

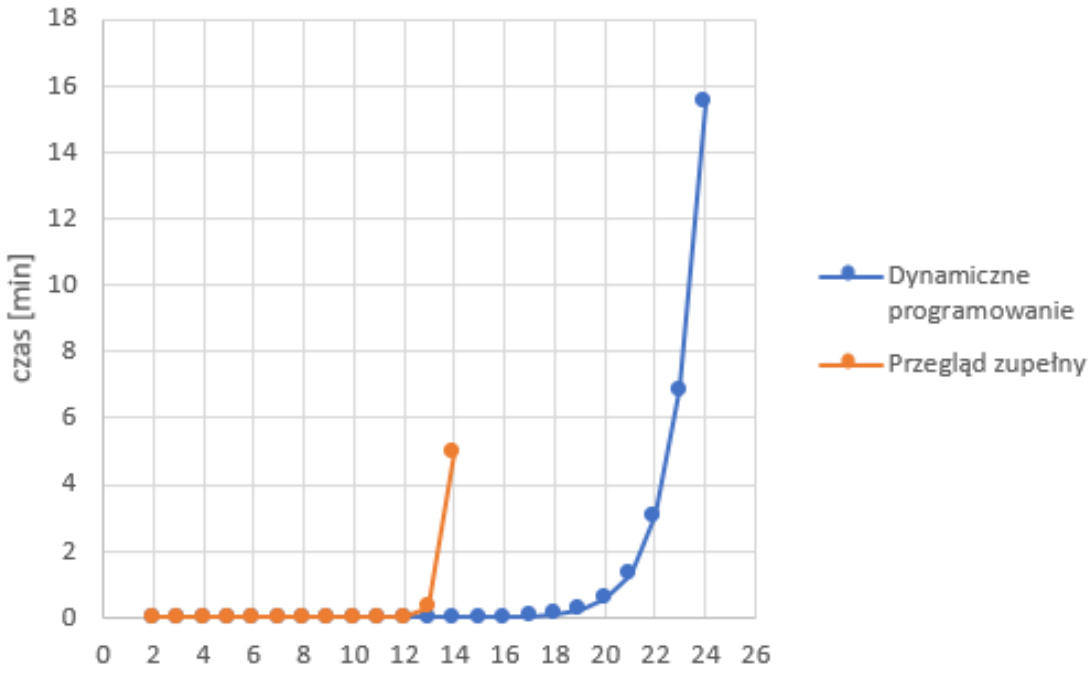
Na wykresie przedstawione są kolejne wyniki pomiarów czasu oraz fragment funkcji *y=x!/100000* dla porównania i oszacowania złożoności czasowej. Dla lepszej widoczności zbieżności wykresów silnia została przemnożona przez stałą a oś jest w skali logarytmicznej.



Na wykresie przedstawione są kolejne wyniki pomiarów czasu oraz fragment funkcji $y = 2^n * n^2/10000$ dla porównania i oszacowania złożoności czasowej. Dla lepszej widoczności zbieżności wykresów złożoności czasowej została przemnożona przez stałą a oś jest w skali logarytmicznej.



W celu uzyskania maksymalnego N dla czasu granicznego, eksperyment został wykonany na jednej instancji dla każdego rozmiaru grafu. Czas graniczny został ustalony na 4 minuty. Jak można odczytać z wykresu oraz tabel, przegląd zupełny przekroczył przeznaczony mu czas już przy 14 wierzchołkach, natomiast algorytm oparty na programowaniu dynamicznym dopiero przy 23.



6 Wnioski

Wyniki eksperymentów nad zaimplementowanymi algorytmami zgadzają się z wiedzą teoretyczną. Złożoność obliczeniowa algorytmu brute force oraz helda-karpa dla asymetrycznego problemu komiwojażera (TSP) zgodnie z przewidywaniem wynoszą O(n!) oraz O($2^n n^2$)). Przy dziesięciu wierzchołkach algorytm helda-karpa okazywał się szybszy od przeglądu zupełnego. Co jest prezentowane na poniższym wykresie podsumowującym.