Projektowanie Efektywnych Algorytmów

Zadanie projektowe nr 1B

Implementacja i analiza efektywności algorytmu podziału i ograniczeń (B&B) oraz programowania dynamicznego (DP) 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 podziału i ograniczeń

Metoda podziału i ograniczeń to heurystyczne podejście, oparte na idei kolejnych podziałów przestrzeni poszukiwań. Kluczowym elementem jest ustalenie dolnego lub górnego ograniczenia kosztu dla dowolnego rozwiązania, zależnie od celu optymalizacji (minimalizacja lub maksymalizacja). Analogicznie do struktury drzewa, traktujemy przestrzeń poszukiwań jako gałęzie, eliminując te, które nie spełniają określonych ograniczeń. Metoda ta pozwala skutecznie zawężać obszar poszukiwań, eliminując niepotrzebne alternatywy i skupiając się na obiecujących ścieżkach rozwiązania problemów optymalizacyjnych.

1.2. Metoda programowania dynamicznego

Programowanie dynamiczne to metoda rozwiązywania problemów, opierająca się na znajdywaniu optymalnych rozwiązań poprzez identyfikację pośrednich punktów pomiędzy aktualnym stanem a zamierzonym celem. W tym podejściu kluczową cechą jest rekurencyjna natura procesu, gdzie każdy kolejny pośredni punkt jest obliczany jako funkcja już rozwiązanych punktów. Dzięki temu programowanie dynamiczne umożliwia unikanie wielokrotnego obliczania tych samych wartości, co przyczynia się do efektywnego rozwiązywania problemów optymalizacyjnych.

1.3. 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.

1.4. Złożoność metody podziału i ograniczeń

W najgorszym przypadku złożoność metody podziału i ograniczeń dla problemu Komiwojażera pozostaje porównywalna do metody przeglądu zupełnego i wynosi O(n!), ponieważ przy zbyt ogólnych ograniczeniach wszystkie możliwe permutacje tras muszą zostać obliczone. W praktyce metoda podziału i ograniczeń wykazuje znakomitą efektywność, zależną od konkretnego przypadku instancji problemu Komiwojażera. Ogólna złożoność czasowa metody zależy przede wszystkim od skuteczności zastosowanej funkcji ograniczającej, która decyduje o liczbie gałęzi podlegających przycięciu. W wielu przypadkach metoda podziału i ograniczeń osiąga złożoność czasową rzędu $O(n^2 * 2^n)$.

1.5. Złożoność metody programowania dynamicznego

Metoda programowania dynamicznego dzięki przechowywaniu wyników obliczeń dla uniknięcia ich wielokrotnego wyliczania często prezentuje znacznie lepszą wydajność niż inne podejścia, zwłaszcza dla mniejszych instancji problemu. Ostateczna złożoność czasowa programowania dynamicznego dla problemu komiwojażera jest zróżnicowana i zależy od konkretnego przypadku oraz implementacji, co umożliwia elastyczne dostosowywanie do różnych wymagań i struktur problemu. W najbardziej efektywnych przypadkach, szacowana złożoność czasowa programowania dynamicznego wynosi $O(n^2 * 2^n)$.

2. Przykład praktyczny

2.1 Metoda podziału i ograniczeń

Zdecydowałem się na implementację metody podziału i ograniczeń, wykorzystując algorytm przeszukiwania w głąb (DFS). W ramach tej implementacji, dla każdego następnika bieżącego wierzchołka, obliczam dolne ograniczenia. Kontynuuję przegląd w węźle, gdzie dolne ograniczenie jest niższe niż ograniczenie górne. Początkowe ograniczenie górne jest wyznaczane za pomocą algorytmu zachłannego. Jeśli znajdę ścieżkę o koszcie niższym niż aktualne ograniczenie górne, aktualizuję je.

Tabela 1. Uśredniony czas wykonania metody programowania dynamicznego dla różnej liczby wierzchołków

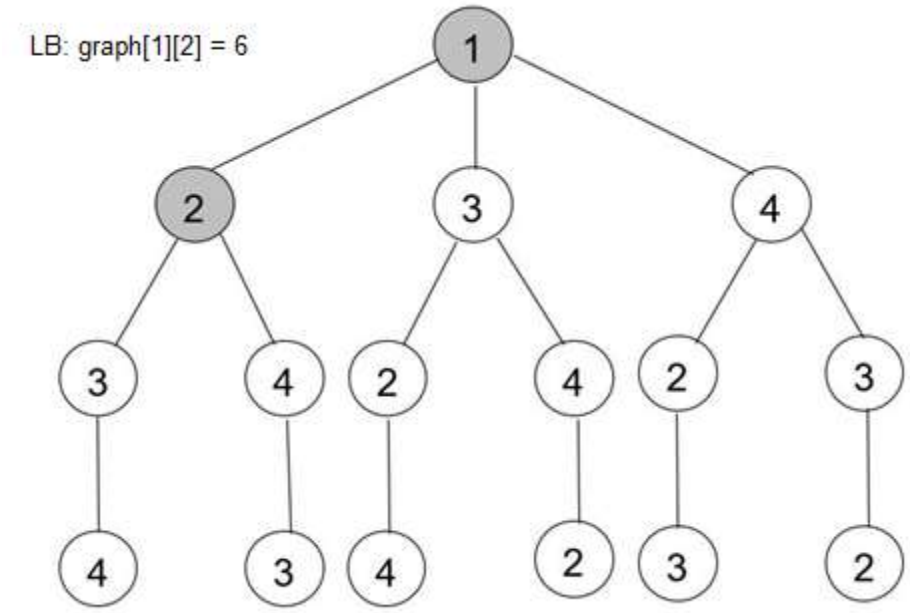
	1	**2**	**3**	**4**
1	-1	6	3	9
2	2	-1	5	4
3	10	8	-1	9
4	5	1	8	-1

Dany jest pełny graf ważony zadany macierzą sąsiedztwa przedstawiony w tabeli 1.

Na podstawie tablicy widzimy, że graf posiada 4 wierzchołki, górne ogrniczenie wyliczone za pomocą algorytmu zachłannego wynosi 20. Ograniczenie dolne na początku wynosi 0.

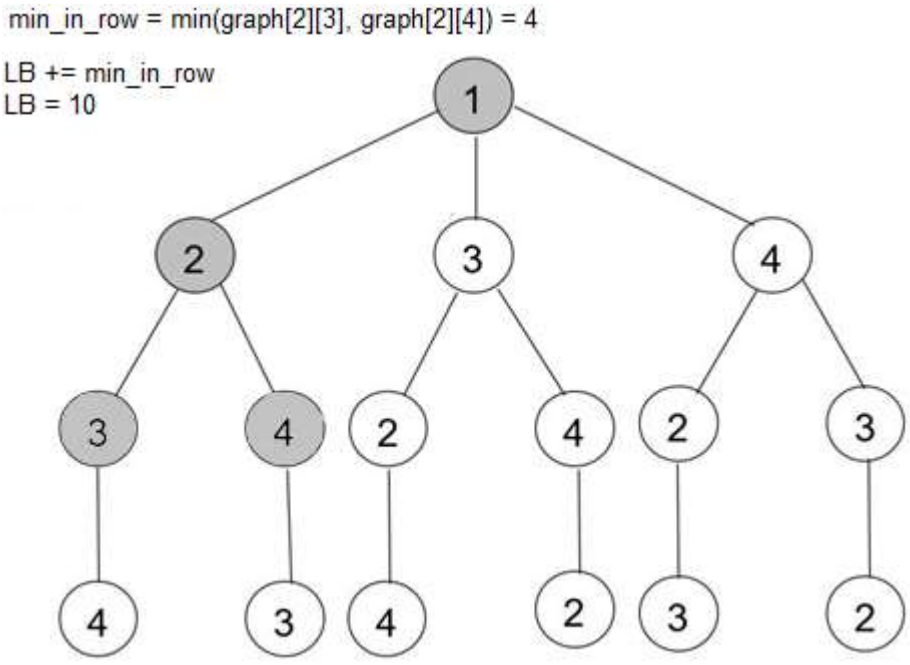
LB = lower_bound (ograniczenie dolne).

Iteracja numer 1:



Rysunek 1. Przedstawia krok pierwszy wykonywany przez algorytm przeszukiwania w głąb.

Algorytm przeszukuje w głąb i wraz z przeszukiwaniem aktualizuje wartość dolnej granicy, w każdej iteracji sprawdza inną gałąź.



Rysunek 2. Przedstawia krok drugi wykonywany przez algorytm przeszukiwania w głąb.

W kolejnym kroku algorytm oblicza ograniczenie dolne dla gałęzi wierzchołka do którego przeszliśmy i porównuje zsumowaną wartość ograniczenia dolnego z ograniczeniem górnym jeżeli ograniczenie dolne jest mniejsze, rekurencyjnie zaczyna algorytm od początku (z ograniczeniem dolnym równym wartości kosztu dotychczasowej ścieżki) dla tych gałęzi aż do odwiedzenia wszystkich wierzchołków, następnie jeżeli koszt obecnej ścieżki jest najmniejszy aktualizuję górne ograniczenie o wartość obecnej ścieżki i zapisuje do zmiennej best_route obecną ścieżkę.

W naszym przypadku po sprawdzeniu ścieżki 1 -> 2 -> 3 -> 4 rekurencyjnie nastąpi wywołanie ścieżki 1 -> 2 -> 4 a po jej obliczeniu zakończy się iteracja 1 i analogicznie algorytm wykona się dla wierzchołków 1 -> 3 w iteracji 2 i 1 -> 4 w iteracji 3.

2.2 Metoda programowania dynamicznego

Programowanie dynamiczne z wykorzystaniem masek bitowych polega na podziale całego problemu na mniejsze podproblemy. Podczas uruchamiania algorytmu dla całego zadania, rekurencyjnie wywołuje się go dla coraz to mniejszych zbiorów wierzchołków, dla których obliczane są koszty. Następnie wyniki uzyskane dla tych mniejszych podproblemów są sumowane, aż do uzyskania kosztu przejścia przez cały graf. Co istotne, algorytm nie analizuje wszystkich możliwych rekurencyjnych podzbiorów, lecz jedynie te, które zbliżają nas do rozwiązania, czyli te, które nie zostały wcześniej obliczone.

Rozważmy przykład prostego grafu o 3 wierzchołkach. Maska oznaczająca odwiedzenie wszystkich wierzchołków wygląda następująco: 111. Pierwsze wywołanie funkcji jest dla argumentów (1, 0), co oznacza, że ruszamy z pierwszego wierzchołka i odwiedziliśmy go. Iterujemy po wszystkich wierzchołkach. Ponieważ pierwszy wierzchołek został odwiedzony, nic nie robimy. Drugi wierzchołek nie był odwiedzony, więc wywołujemy tę funkcję rekurencyjnie dla argumentów (011, 1), dodajemy wagę krawędzi i zapisujemy wynik w tablicy dp_memory[3][1]. Ponownie iterujemy. Ponieważ pierwszy i drugi wierzchołek były odwiedzone, nic nie robimy. Trzeci wierzchołek nie był odwiedzony, więc ponownie wywołujemy funkcję dla argumentów (111, 2). Maska wskazuje, że odwiedziliśmy wszystkie wierzchołki, więc kończymy rekurencję, pamiętając o dodaniu

wagi krawędzi między ostatnim a pierwszym wierzchołkiem. Zapisujemy wynik w tablicy dp_memory a w tablicy parents_memory zapisujemy ścieżkę optymalną bazując na dotychczasowych porównaniach kosztów ścieżki dla podanej maski i obecnego wierzchołka. Następnie wracamy do iteracji w pierwszym wywołaniu funkcji i kontynuujemy postępowanie analogicznie.

3. Opis implementacji algorytmu

3.1 Metoda podziału i ograniczeń

vector< vector<*int*>> graph – macierz dynamiczna przechowująca połączenia między miastami w formie grafu
vector<*int*> vertexes – tablica dynamiczna przechowująca wszystkie wierzchołki grafu
vector<*int*> upper_bound_path – tablica dynamiczna przechowująca obecną ścieżkę górnego ograniczenia
vector<*int*> best_route – tablica dynamiczna przechowująca ścieżkę o najmniejszym koszcie
int graph_size – zmienna typu int przechowująca ilość wierzchołków grafu
int upper_bound_value – zmienna typu int przechowująca wartość ograniczenia górnego

Klasa składa się z przeciążonego konstruktora oraz siedmiu metod:

- Konstruktor przypisuje wartość do zmiennej graph, przekazywany argument to macierz miast i tras pomiędzy nimi. Następnie w zmiennej graph_size zostaje przypisana ilość miast. Kolejna linijka wywołuje metode: available_vertexes() która inicjalizuje zmienna vertexes wszystkimi wierzchołkami w grafie. Na koniec zostaje wywołana metoda perform_upper_bound() która inicjalizuje zmienna upper_bound_value z wartością obliczonego przez metode górnego ogranicznia.
- Metoda void available_vertexes()

  Metoda inicjalizuje tablice dynamiczną vertexes wszystkimi wierzchołkami w grafie.

- Metoda int perform_upper_bound()

  Metoda wywołuje metodę greed_method() z parametrami: wiersz = 0 i tablicą vertexes która zwraca wartość ograniczenia górnego, następnie metoda zwraca tą wartość.

- Metoda int greedy_method(int row, std::vector<int> unvisited_vertexes)

  Metoda oblicza ograniczenie górne algorytmem zachłannym.

```
int BnB::greedy_method(int row, std::vector<int> unvisited_vertexes) {

    if (unvisited_vertexes.size() == 1) {
        upper_bound_path.push_back(0);
        return graph[row][0];
    }
    int lowest_cost = INT_MAX;
    int index;
    for (int i = 0; i < graph.size(); i++) {
        if ((lowest_cost > graph[row][i]) && row != i) {
            auto it : iterator<...> = std::find( first: unvisited_vertexes.begin(), last: unvisited_vertexes.end(), val: i);
            if (it != unvisited_vertexes.end()) {
                lowest_cost = graph[row][i];
                index = i;
            }

        }
    }
    upper_bound_path.push_back(index);
    unvisited_vertexes.erase( first: std::remove( first: unvisited_vertexes.begin(), last: unvisited_vertexes.end(), value: row),
                              last: unvisited_vertexes.end());
    lowest_cost += greedy_method( row: index, unvisited_vertexes);
    return lowest_cost;
}
```

Rysunek 3. Przedstawiający kod metody greedy_method()

Pierwszy warunek sprawdza czy został tylko jeden nieodwiedzony wierzchołek, jeżeli tak to zwraca wartość ścieżki z niedowiedzonego wierzchołka do wierzchołka początkowego.

W linii 26 i 27 rysunku 3 następuje inicjalizacja zmiennych do śledzenia kosztu i indeksu następnego wierzchołka.

Pętla iteruje po każdym wierzchołku grafu, warunek w pętli sprawdza czy krawędź wierzchołka jest mniejsza od wartości najmniejszej krawędzi dotychczas znalezionej i czy nie jest sprawdzana wartość z wierzchołka do samego siebie.

Jeżeli oba warunki są spełnione i wierzchołek nie był jeszcze odwiedzany to następuje przypisanie zmiennej lowest_cost o wartość najmniejszej krawędzi i zmiennej index na mu odpowiadający.

Następnie do tablicy upper_bound_path zostaje dodany wierzchołek do ścieżki a z tablicy unvisited_vertexes zostaje usunięty ten wierzchołek.

Na koniec do zmiennej lowest_cost dodajemy zwrócona wartość przez rekurencyjne wywołanie metody greedy_method() z index odpowiadający obecnemu wierzchołkowi w którym się znajdujemy i zaktualizowaną tablicą unvisited_vertexes.

Metoda zwraca wartość lowest_cost.

- Metoda int distance(vector<int> route, int route_size)

  Metoda oblicza całkowitą odległośc trasy poprzez zsumowanie odległości pomiędzy kolejnymi wierzchołkami danej trasy.

- Metoda vector<int> bnb_dfs()

  Metoda inicjalizuję tablicę vertices_to_check o wierzchołki które nie zostały zwiedzone następnie wywołuje bnb_dfs_recursion() z tablicą vertices_to_check oraz tablica route która jest pusta.

  Na koniec zwraca wartość zmiennej best_route.

- Metoda void bnb_dfs_recursion(vector<int> vertices_to_check, vector<int> current_route)

  Ta funkcja przyjmuje jako dane wejściowe dwa wektory: vertices_to_check (pozostałe wierzchołki do zbadania) i current_route (badana bieżąca trasa częściowa).

```
void BnB::bnb_dfs_recursion(std::vector<int> vertices_to_check, std::vector<int> current_route) {

    std::vector<int> next_route = current_route;
    std::vector<int> next_vertices_to_check = vertices_to_check;

    int lower_bound=0, min_in_row=0;

    if (vertices_to_check.size() > 0) {...}
    else {

        lower_bound = distance( route: current_route,  route_size: current_route.size());

        if (lower_bound < upper_bound_value) {
            upper_bound_value = lower_bound;
            best_route = current_route;
        }
    }

}
```

Rysunek 4. Przedstawiający fragment kodu metody bnb_dfs_recursion()

Warunek główny z rysunku 4 jeśli nie ma więcej wierzchołków do sprawdzenia (vertices_to_check.size() == 0), oblicz odległość bieżącej trasy i zaktualizuj wartości Upper_bound_value i best_route, jeśli zostanie znalezione lepsze rozwiązanie.

W przeciwnym wypadku:

```
if (vertices_to_check.size() > 0) {
    for (int i = 0; i < vertices_to_check.size(); i++) {
        next_route.push_back(vertices_to_check[i]);
        next_vertices_to_check.erase( position: next_vertices_to_check.begin() + i);


        for (int j = 0; j < next_route.size() - 1; j++) {
            lower_bound += graph[next_route[j]][next_route[j + 1]];
        }
```

Rysunek 5. Przedstawiający fragment kodu metody bnb_dfs_recursion(). Linijki 79 - 87

Segment kodu z rysunku 5 bada różne trasy, wybierając wierzchołki z vertices_to_check i usuwa go z next_vertices_to_check, upewniając się, że ten sam wierzchołek nie będzie ponownie brany pod uwagę w tej samej trasie

W kolejnej pętli oblicza częściowy koszt odległości, sumując odległości pomiędzy kolejnymi wierzchołkami w next_route (oblicza trasę dotychczas zwiedzoną)

```
if (next_vertices_to_check.size() > 0) {
    min_in_row = graph[next_route[next_route.size() - 1]][next_vertices_to_check[0]];

    for (int j = 0; j < vertices_to_check.size(); j++) {
        if ((min_in_row > graph[next_route[next_route.size() - 1]][vertices_to_check[j]]) &&
        (graph[next_route[next_route.size() - 1]][vertices_to_check[j]] != -1)) {
            min_in_row = graph[next_route[next_route.size() - 1]][vertices_to_check[j]];
        }
    }

    lower_bound += min_in_row;
}
```

Rysunek 6. Przedstawiający fragment kodu metody bnb_dfs_recursion(). Linijki 90 - 101

Pierwszy warunek na rysunku 6 oznacza, że jeżeli istnieje wierzchołek nieodwiedzony dla danej ścieżki przypisuje wartość pierwszego wierzchołka jeszcze nieodwiedzonego z tablicy next_verices_to_check do zmiennej min_in_row po czym następuje pętla w celu znalezienia najmniej kosztownej krawędzi do możliwych wierzchołków nieodwiedzonych. Ten blok kodu zasadniczo polega na znalezieniu minimalnej odległości od ostatniego wierzchołka bieżącej trasy do dowolnego wierzchołka w wektorze vertices_to_check. Informacje te są następnie wykorzystywane do aktualizacji zmiennej lower_bound w całym algorytmie.

```
for (int j = 0; j < next_vertices_to_check.size(); j++) {
    min_in_row = graph[next_vertices_to_check[j]][0];
    for (int k = 0; k < next_vertices_to_check.size(); k++) {
        if ((graph[next_vertices_to_check[j]][next_vertices_to_check[k]] < min_in_row) &&
        graph[next_vertices_to_check[j]][next_vertices_to_check[k]] != -1) {
            min_in_row = graph[next_vertices_to_check[j]][next_vertices_to_check[k]];
        }
    }
    lower_bound += min_in_row;
}

if (lower_bound < upper_bound_value) {
    bnb_dfs_recursion( vertices_to_check: next_vertices_to_check,  current_route: next_route);
}


next_route = current_route;
next_vertices_to_check = vertices_to_check;
lower_bound = 0;
```

Rysunek 7. Przedstawiający fragment kodu metody bnb_dfs_recursion(). Linijki 103 - 121

Fragment kodu na rysunku 7 znajduje minimalną odległość od j-tego wierzchołka w next_vertices_to_check zarówno do wierzchołka początkowego (wierzchołek 0), jak i dowolnego innego wierzchołka w next_vertices_to_check jeżeli są w nim wierzchołki nieodwiedzone. Wartość najmniejszą przypisuje do zmiennej min_in_row która następnie dodawana jest do zmiennej lower_bound.

Ostatni warunek na rysunku 7 sprawdza czy wartość ograniczenia dolnego jest mniejsza od wartości ograniczenia górnego, jeżeli tak kontynuuje algorytm poprzez rekurencyjne wywołanie samej siebie z zaktualizowanymi wartościami tablicy next_vertices_to_check oraz tablicy next_route.

- Metoda void show_lowest_path()

  Metoda służy do wyświetlenia w konsoli najmniej kosztownej ścieżki oraz jej wartości.

3.2 Metoda programowania dynamicznego

vector< vector<int>> matrix – macierz dynamiczna przechowująca połączenia między miastami w formie grafu

int instance_size - ilość miast

vector < vector <int> > dp_memory – macierz dynamiczna o rozmiarze $[2^n][n]$ służąca zapamiętywaniu obliczonych już rozwiązań

vector < vector <int> > parents_memory – macierz dynamiczna o rozmiarze $[2^n][n]$ służąca do wyciągania samej ścieżki za pomocą backtracking (przeszukiwanie wsteczne).

vector<int> route - tablica dynamiczna służąca do tymczasowego przechowania wartości niezbędnych do inicjalizacji macierzy dp_memory i parents_memory

int visited_all – maska bitowa reprezentująca wszystkie miasta jako odwiedzone

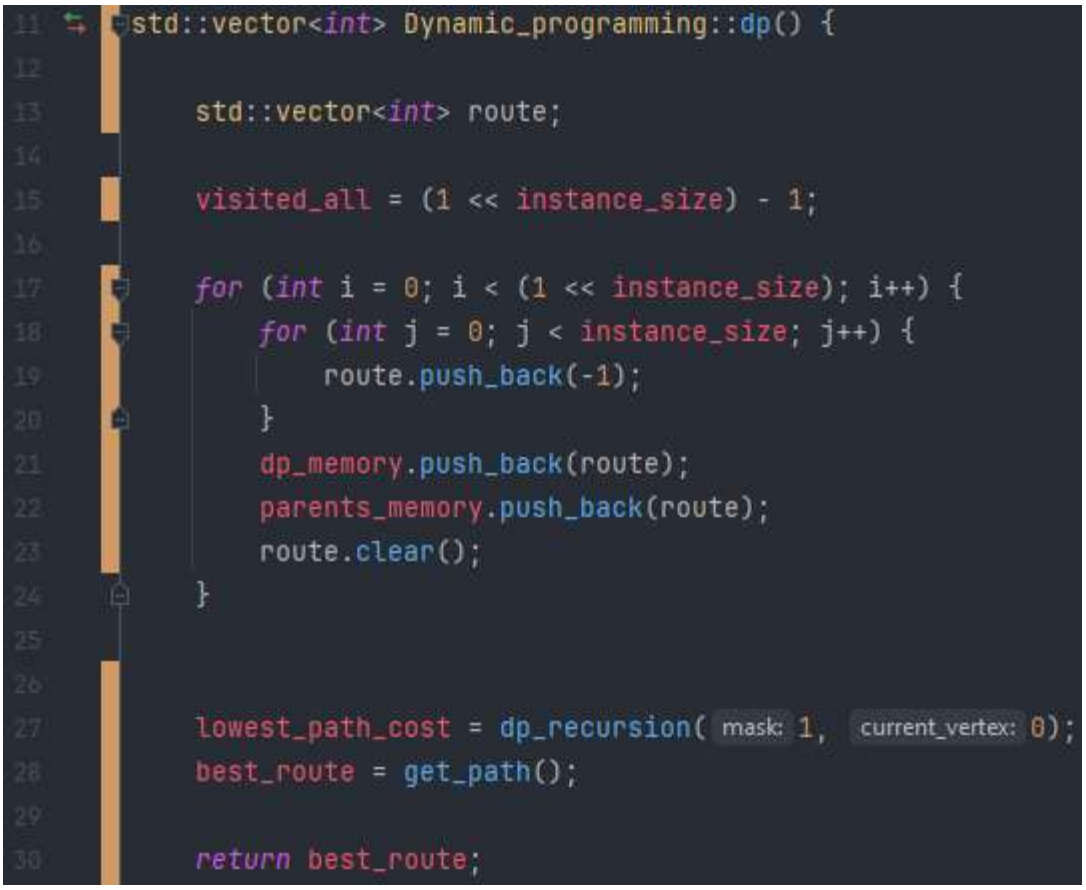
int lowest_path_cost – zmienna przechowująca koszt ścieżki o najmniejszym koszcie

vector<int> best_route – tablica dynamiczna przechowująca ścieżkę o najmniejszym koszcie

int IMAX – zmienna wykorzystywana do porównań z kosztem obecnej ścieżki, zaincjalizowana z wartościa = 2147483647

Klasa składa się z przeciążonego konstruktora oraz pięciu metod:

- Konstruktor przypisuje wartość do zmiennej matrix, przekazywany argument to macierz miast i tras pomiędzy nimi. Następnie w zmiennej instance_size zostaje przypisana ilość miast.
- Metoda vector<int> dp()



Rysunek 8. Przedstawiający metodę dp()

Pętle for na rysunku 8 służą do inicjalizacji macierzy dp_memory i parents_memory, wartość -1 oznacza, że odpowiadający mu podproblem nie został jeszcze obliczony.

Dla zmiennej lowset_path_cost następuje wyłowanie metody dp_recursion z parametrami mask = 1 i current_vertex = 0 które znaczą odwiedź miasto 1 i na taką wartość też przypisz maskę a obecne miasto w którym się znajdujesz to miasto 0.

Zmienna best_route zostaje obliczona z wylowaniem metody get_path() która służy do przeszukania wstecznego aby wyszukać optymalną ścieżkę odpowiadającą minimalnemu kosztowi.

Metoda zwraca vector przechowujący ścieżkę o najmniejszym koszcie.

- Metoda int dp_recursion(int mask, int current_vertex)

```
int Dynamic_programming::dp_recursion(int mask, int current_vertex) {

    if (mask == visited_all) {
        return matrix[current_vertex][0];
    }

    if (dp_memory[mask][current_vertex] != -1) {
        return dp_memory[mask][current_vertex];
    }

    int cost = IMAX;

    for (int city = 0; city < instance_size; city++) {

        if ((mask & (1 << city)) == 0) {

            int current_cost = matrix[current_vertex][city] + dp_recursion( mask: mask | (1 << city),  current_vertex: city);
            cost = std::min(cost, current_cost);

            if (current_cost == cost) {
                dp_memory[mask][current_vertex] = cost;
                parents_memory[mask][current_vertex] = city;
            }
        }
    }

    return cost;
}
```

Rysunek 9. Przedstawiający metodę dp_recursion()

Pierwszy warunek w metodzie przedstawiony na rysunku 9 sprawdza czy wszystkie miasta zostały odwiedzone jeżeli tak to zwraca koszt powrotu z obecnego miasta do miasta początkowego

Drugi warunek sprawdza czy obecny podproblem został rozwiązany w przeszłości, jeżeli tak, zwraca wcześniej obliczony koszt dla podproblemu

Następnie zaczyna się pętla odpowiedzialna za odwiedzenie wszystkich nieodwiedzonych miast z obecnego miasta.

Dla każdego miasta oblicza koszt zwiedzenia go dodając z wywołaniem rekurencyjnie samej siebie z parametrami maski która posiada obecne miasto jako odwiedzone i current_vertex równy obecnemu miastu.

Wynik tej sumy jest porównywany z obecnie najmniej kosztująca ścieżką, jeżeli jest mniejsze to ścieżka i jej koszt jest zapisywana w zmiennych.

Metoda zwraca koszt ścieżki o najmniejszym koszcie

- Metoda vector<int> get_path()

  Metoda ta wykorzystuje informacje przechowywane w macierzy parents_memory do prześledzenia od stanu końcowego do stanu początkowego, konstruując optymalną ścieżkę

- Metoda void show_lowest_path()

  Metoda wypisuje w konsoli ścieżkę o najmniejszym koszcie
- Metoda string get_path_string()

  Metoda zwraca vector przechowujący ścieżkę o najmniejszym koszcie

4. Plan eksperymentu

W celu zapewnienia miarodajnej analizy implementacji algorytmu, został zmierzony czas dla grafów od 10 do 24 wierzchołków gdzie dla każdego rozmiaru grafu zostało przeprowadzonych 100 pomiarów czasowych losowych instancji problemu których wyniki pomiaru uśredniono.

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 procesorze Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz z 32,0 GB pamięci RAM o szybkości 2133 MHz i systemie operacyjnym Windows 10 Education N 21H2.

Wykresy zostały wygenerowane w oprogramowaniu Microsoft Excel.

5. Wyniki eksperymentów

Tabela 2. Uśredniony czas wykonania metody programowania dynamicznego dla różnej liczby wierzchołków

Liczba wierzchołków	Czas [μs]	Czas [s]
10	228,2	2,28E-04
11	496,89	4,97E-04
12	1053,74	1,05E-03
13	2400,1	2,40E-03
14	5304,32	5,30E-03
15	11368,75	1,14E-02
16	27459,45	2,75E-02
17	72454,52	7,25E-02
18	178948,55	1,79E-01
19	444453,26	4,44E-01
20	1098670,46	1,10E+00
21	2734947,58	2,73E+00
22	6491837,68	6,49E+00
23	14936845,2	1,49E+01
24	34773471,1	3,48E+01

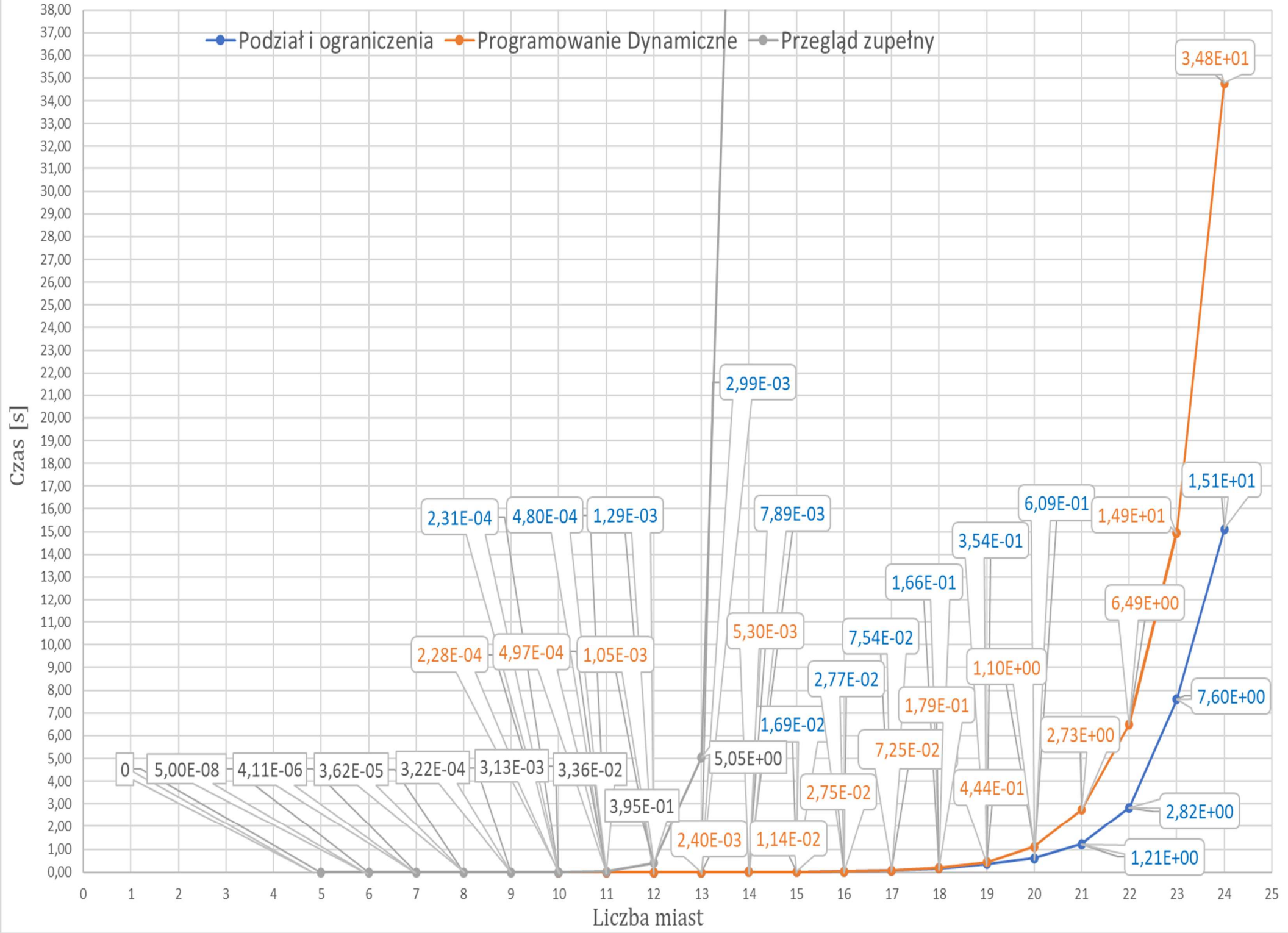
Uzyskany uśredniony czas w tabeli 2 świadczy o prawidłowej implementacji algorytmu i jest bliski oczekiwanemu czasowi dla programowania dynamicznego. Brak pomiarów dla większej liczby wierzchołków spowodowany jest zbyt długim czasem oczekiwania na wyniki.

Tabela 3. Uśredniony czas wykonania metody podziału i ograniczeń dla różnej liczby wierzchołków

Liczba wierzchołków	Czas [μs]	Czas [s]
10	231,01	2,31E-04
11	480,03	4,80E-04
12	1287,88	1,29E-03
13	2986,48	2,99E-03
14	7891,02	7,89E-03
15	16947,71	1,69E-02
16	27742,72	2,77E-02
17	75424,08	7,54E-02
18	166399,07	1,66E-01
19	354166,22	3,54E-01
20	608879,06	6,09E-01
21	1213881,6	1,21E+00
22	2821346,2	2,82E+00
23	7595975,82	7,60E+00
24	15092712,6	1,51E+01

Wyniki uśrednionych pomiarów z tabeli 3 znajdują się w zakresie wartości oczekiwanych teoretycznie, świadczą o efektywnej implementacji i poprawnym wykonywaniu się algorytmu. Pomiary do 24 wierzchołków są wynikiem długiego czasu oczekiwania na wyniki dla większej ich liczby.

Wykres 1. Zależność czasu wykonania metod od rozmiaru problemu



Zgodnie z teorią na wykresie 1 dla instancji problemu od 20 wierzchołków metoda podziału i ograniczeń jest optymalna natomiast dla mniejszych instancji metoda programowania dynamicznego. Spowodowane jest to efektywnością ograniczeń które znacznie zmniejszają obszar eksploracji co jest znaczące szczególnie dla większych instancji problemu.

6. Wnioski

Wyniki pomiarów dla obu metod znajdują się w oczekiwanych przedziałach wartości teoretycznych, jest to pozytywny sygnał dotyczący jakości algorytmów.

Implementację obu metod wykonują się poniżej , szacowanej złożoności czasowej $O(n^2 * 2^n)$ dla instancji badanych pomiarowo, sugeruje to, że są one dobrze zoptymalizowane i sprawdzają się w praktyce.

Wraz ze wzrostem rozmiaru problemu, metoda podziału i ograniczeń może skutecznie ograniczyć obszar przeszukiwania, co prowadzi do szybszego znalezienia optymalnego rozwiązania. Natomiast dla mniejszych instancji, gdzie przestrzeń poszukiwań nie jest tak duża, metoda programowania dynamicznego przeważnie jest bardziej wydajna.

Warto zauważyć, że czas przygotowania danych wejściowych dla metody programowania dynamicznego może być znacznie krótszy niż dla metody podziału i ograniczeń, czynniki decydujące o wyborze między metodą programowania dynamicznego a metodą podziału i ograniczeń obejmuje nie tylko rozmiar problemu, ale także charakterystykę danych wejściowych i dostępność zasobów obliczeniowych.

Analiza wykresu 1 jednoznacznie potwierdza wykładniczą złożoność czasową algorytmu przeglądu zupełnego. W praktyce oznacza to, że metoda ta akceptowalna jest pod względem czasu wykonania jedynie dla małych instancji problemu komiwojażera.