Programowanie Efektywnych Algorytmów

Sprawozdanie z zadania projektowego numer 1 - ***Implementacja i analiza efektywności algorytmu podziału i ograniczeń i programowania dynamicznego.***

Autor: Krzysztof Krajewski 272877

Wprowadzenie

Problem komiwojażera w wersji asymetrycznej (ATSP, ang. Asymmetric Traveling Salesman Problem) jest wariantem klasycznego problemu komiwojażera (TSP, ang. Traveling Salesman Problem), w którym koszty przejścia między dwoma miastami nie są symetryczne, tzn. koszt przejścia z miasta A do miasta B może różnić się od kosztu przejścia z miasta B do miasta A. Formalnie, problem można opisać za pomocą skierowanego grafu pełnego G = (V, E), gdzie V to zbiór wierzchołków (miast), a E to zbiór skierowanych krawędzi o przypisanych kosztach. Celem jest znalezienie cyklu Hamiltona, który minimalizuje łączny koszt podróży i odwiedza każde miasto dokładnie raz.

Złożoność obliczeniowa problemu ATSP

ATSP jest problemem NP-trudnym, co oznacza, że nie jest znany algorytm, który mógłby rozwiązać go w czasie wielomianowym dla dowolnej liczby miast. Rozmiar przestrzeni rozwiązań rośnie wykładniczo wraz z liczbą miast N, ponieważ liczba możliwych permutacji wynosi $(n-1)!$. Z tego względu optymalne rozwiązania dla ATSP są trudne do uzyskania dla dużych instancji problemu. W tym sprawozdaniu przeprowadzimy analizę następujących podejść do rozwiązania tego problemu:

- Przegląd zupełny (ang. Brute Force)
- Metoda rozgałęzień i ograniczeń (ang. Branch and Bound)
- Programowanie dynamiczne (ang. Dynamic Programming)

Omówienie algorytmów

Wszystkie przedstawione algorytmy działają z użyciem klasy CityMatrix, która zawiera dwuwymiarową tablicę przechowującą macierz odległości pomiędzy miastami wraz z metodami obsługującymi jej inicjalizację, generację i wczytywanie.

Brute Force

Najprostszy algorytm rozwiązujący problem ATSP poprzez sprawdzenie wszystkich możliwych rozwiązań i wybranie tego o najmniejszym koszcie. Jego złożoność czasowa to O(n!), co sprawia, że jest on praktyczny tylko dla niewielkich instancji problemu. Zaimplementowany algorytm w metodzie TSP::bruteForceNew rozwiązuje problem poprzez iteracyjne sprawdzanie wszystkich możliwych permutacji tras między miastami. Początkowo algorytm inicjalizuje tablicę path, która reprezentuje ścieżkę odwiedzin miast, zaczynając od miasta 0 i kończąc na nim po odwiedzeniu wszystkich pozostałych. Ustawia zmienną min_cost na bardzo wysoką wartość, aby później móc porównywać koszty. Następnie, algorytm generuje kolejne permutacje trasy przy użyciu funkcji nextPermutation i

oblicza koszt każdej z nich za pomocą funkcji calculatePathCost. Jeśli koszt aktualnej trasy jest niższy od dotychczasowego minimum, aktualizuje min_cost oraz zapisuje tę trasę jako najlepszą w tablicy best_path. Po sprawdzeniu wszystkich możliwych permutacji algorytm wypisuje minimalny koszt oraz optymalną ścieżkę na standardowe wyjście, prezentując rozwiązanie problemu.

Kod 1. Implementacja algorytmu Brute Force.

```
void TSP::bruteForceNew(cityMatrix &city_matrix) {
    int num_cities = city_matrix.size;
    int path[num_cities + 1];
    int min_cost = INT_MAX;
    int best_path[num_cities];

    // Initialize the path with the first permutation
    for (int i = 0; i < num_cities; i++) {
        path[i] = i;
    }
    path[num_cities] = 0; // Return to the starting city

    // Generate permutations and find minimum cost path iteratively
    do {
        // Calculate the cost of the current path
        int current_cost = calculatePathCost(city_matrix, path);
        if (current_cost < min_cost) {
            min_cost = current_cost;
            for (int i = 0; i < num_cities; i++) {
                best_path[i] = path[i]; // Update the best path
            }
        }

    } while (nextPermutation(path, num_cities));

    // Output the best path and its cost
    std::cout << "Minimum cost: " << min_cost << std::endl;
    std::cout << "Path: ";
    std::cout << "0 ";
    for (int i = 1; i < num_cities; i++) {

        std::cout << "-> " << best_path[i] << " ";
    }
    std::cout << "-> 0"; // add the starting city to the end
    std::cout << std::endl;

}
```

Algorytm Branch & Bound

Jest to metoda przeszukiwania, która redukuje przestrzeń poszukiwań poprzez odrzucanie pewnych podproblemów na podstawie oszacowań dolnego ograniczenia kosztu. Choć jego złożoność teoretyczna w najgorszym przypadku wynosi również O(n!), w praktyce działa on szybciej niż brute force (szczególnie przy N > 15), dzięki redukcji liczby przetwarzanych gałęzi drzewa wyszukiwań. Zaimplementowany algorytm Branch & Bound w metodzie TSP::branchAndBound, działa poprzez rozgałęzianie i ograniczanie przestrzeni rozwiązań, aby znaleźć optymalną ścieżkę o najniższym koszcie. Algorytm rozpoczyna od utworzenia węzła głównego (korzenia), który zawiera zredukowaną macierz kosztów oraz ścieżkę startową od miasta 0. Oblicza się w nim początkowy koszt, czyli dolną granicę, używając redukcji macierzy.

Funkcja obliczająca ograniczenia (reduceMatrix) służy do wyznaczania dolnej granicy kosztów dla danego węzła. Jej zadaniem jest zredukowanie wartości w macierzy kosztów poprzez minimalizację poszczególnych wierszy i kolumn. Proces działania funkcji polega na iteracyjnym przechodzeniu przez każdy wiersz macierzy i znalezieniu najmniejszej wartości, którą następnie odejmuje od wszystkich elementów w danym wierszu. Podobnie postępuje się z kolumnami: dla każdej kolumny znajduje się najmniejszą wartość i odejmuje od wszystkich jej elementów. Po zakończeniu redukcji sumuje się minimalne wartości dla wszystkich wierszy i kolumn, aby uzyskać całkowity koszt redukcji. Funkcja ta pozwala na obliczenie dolnej granicy kosztu dla nowo utworzonego węzła, co z kolei umożliwia określenie, czy węzeł powinien być dodany do kolejki do dalszej eksploracji. Koszt węzła stworzony m. in. przy użyciu kosztu redukcji mówi jaki minimalny koszt można uzyskać podążając daną ścieżką. Na przykład dla ścieżki 0-1-2-3 koszt dla węzła trzeciego wynosi 56, oznacza to, że dla ścieżki 0-1-2-3[…] (gdzie […] oznacza dowolną kombinację węzłów dozwoloną jako rozwiązanie) minimalny koszt jaki można uzyskać to 56. Dzięki temu algorytm unika przeszukiwania ścieżek, które już na początku wydają się kosztowne, co znacząco redukuje przestrzeń przeszukiwania i przyspiesza proces znajdowania rozwiązania.

W kolejnych krokach algorytm przechodzi do eksploracji możliwych ścieżek poprzez tworzenie nowych węzłów dla każdej potencjalnej krawędzi prowadzącej do następnego miasta. Każdy nowy węzeł zawiera kopię macierzy kosztów, zaktualizowaną o ograniczenia wynikające z wybranego przejścia między miastami oraz koszt ścieżki uwzględniający sumę dotychczasowego kosztu, kosztu przejścia i kosztu redukcji macierzy. Jeśli koszt nowego węzła jest mniejszy niż obecnie najlepszy znaleziony koszt (górna granica), węzeł zostaje dodany do kolejki priorytetowej. W przeciwnym razie jest odrzucany, co pozwala ograniczyć przestrzeń przeszukiwania.

Algorytm kontynuuje eksplorację do momentu, aż znajdzie optymalną trasę. Na koniec zwalnia pamięć zajmowaną przez macierze dla przetworzonych węzłów i wypisuje wynik: minimalny koszt oraz optymalną trasę odwiedzin miast.

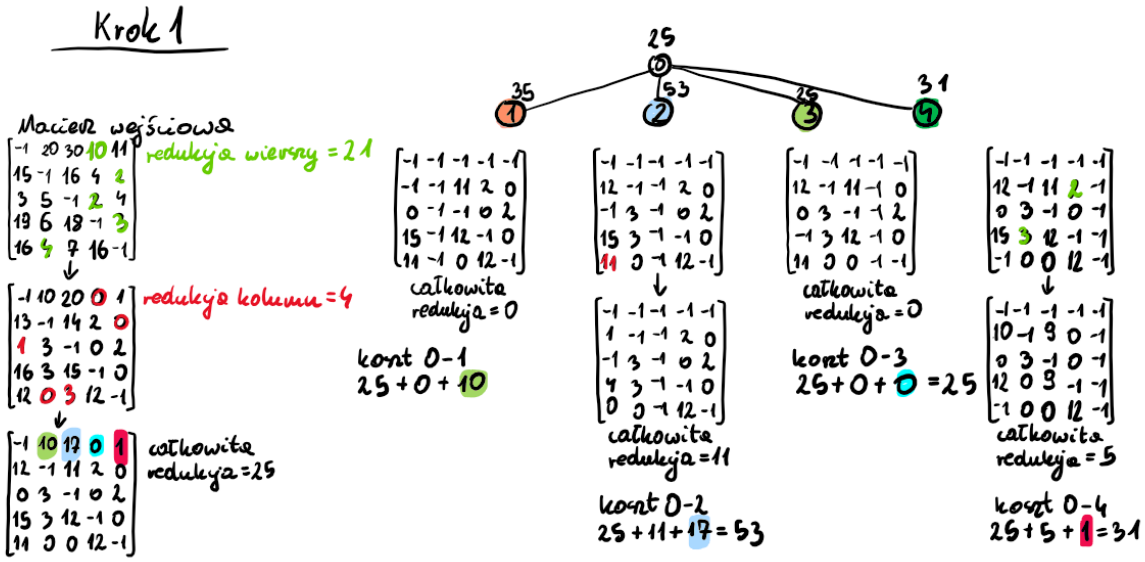
Przykład działania algorytmu:

Na rysunku 1 widać pierwszy krok algorytmu – obliczenie kosztów dla każdego węzła połączonego z węzłem zerowym. Macierz wejściowa jest macierzą, którą program otrzymuje z klasy CityMatrix. Macierz ta zostaje zredukowana poprzez znalezienie minimalnych wartości w każdym wierszu i kolumnie oraz odjęcie ich od całego wiersza i całej kolumny. Po tym procesie otrzymujemy macierz zredukowaną dla węzła zerowego oraz koszt węzła zerowego wynoszący w tym wypadku 25. Macierz zredukowana posłuży następnie do generacji macierzy zredukowanych dla każdego innego węzła (1, 2, 3 oraz 4). Aby tego dokonać, dla każdej pary (0-1, 0-2, 0-3, 0-4) wykonywane będą redukcje zredukowanej wcześniej macierzy zmodyfikowanej w taki sposób, że:

- Wiersz odpowiadający pierwszemu węzłowi z pary (w tym wypadku 0) zamieniony zostanie na wartość -1,
- Kolumna odpowiadająca drugiemu węzłowi z pary (1, 2, 3 lub 4) zamieniona zostanie na wartość -1,
- Aby zapobiec zbyt wczesnemu powrotowi do węzła zerowego wartości odpowiadające drodze z wierzchołka, dla którego aktualnie redukowana jest macierz do wierzchołka o numerze 0 zamieniona zostanie na wartość -1.

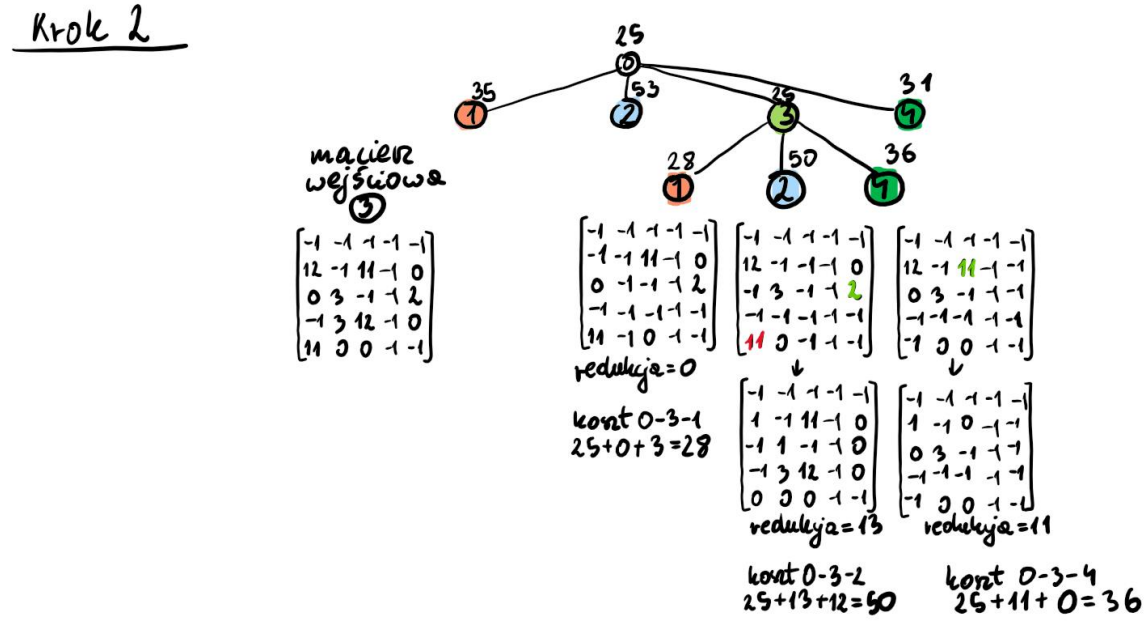
Po dokonaniu redukcji zostanie obliczony koszt dla każdego węzła ze wzoru: **koszt węzła rodzica + koszt całkowitej redukcji + koszt ścieżki między parą węzłów zgodnie z macierzą rodzica**. Na przedstawionym przykładzie kosztem węzła rodzica jest 25 – bo taki koszt ma węzeł 0, koszty całkowitej redukcji znajdują się pod macierzami każdego z węzłów, a koszty ścieżek zaznaczone są odpowiadającymi kolorami na macierzy powiązanej z węzłem 0.

Uwaga. Wiersze i węzły ustawione na -1 są traktowane jako zredukowane.

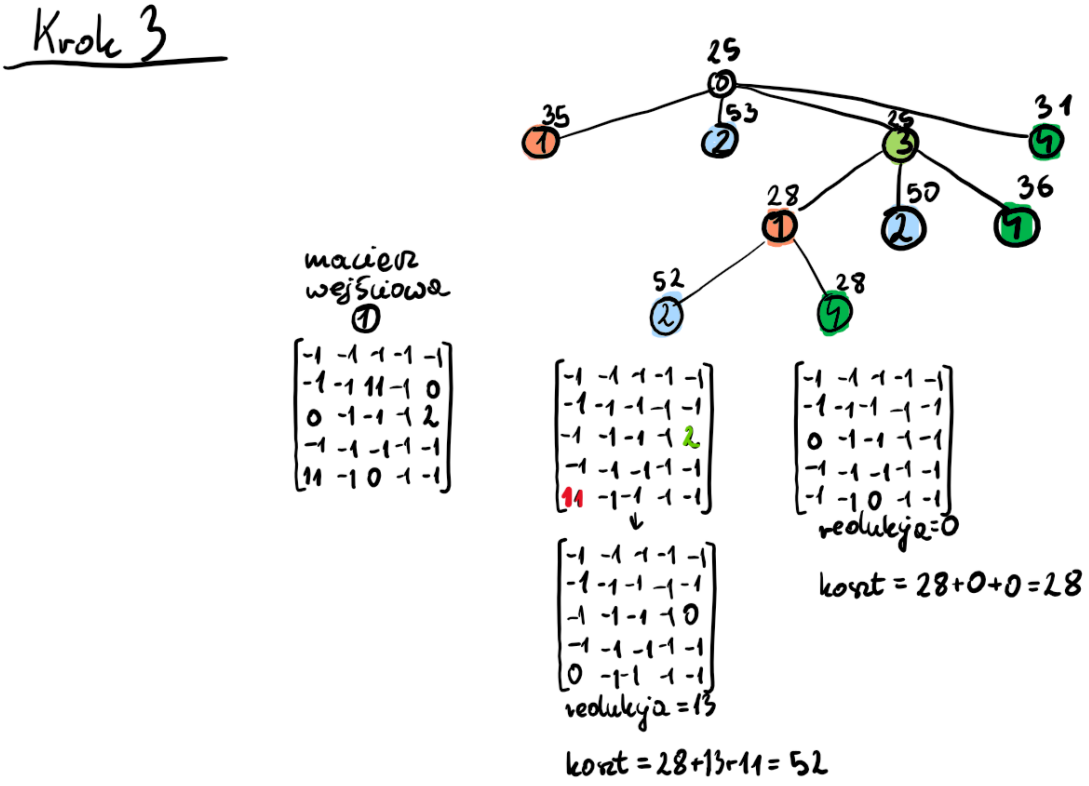


Rysunek 1. Krok pierwszy algorytmu - pierwsze rozwinięcie przestrzeni rozwiązań.

Drugi krok algorytmu przedstawiony został na rysunku 2. Następnym rozpatrywanym węzłem jest ten o najniższym koszcie – w tym przypadku węzeł 3. Macierzą wejściową będzie teraz macierz powiązana właśnie z węzłem trzecim. Dla kolejnych par węzłów (3-1, 3-2, 3-4) wykonywane są redukcje i obliczane są koszty.



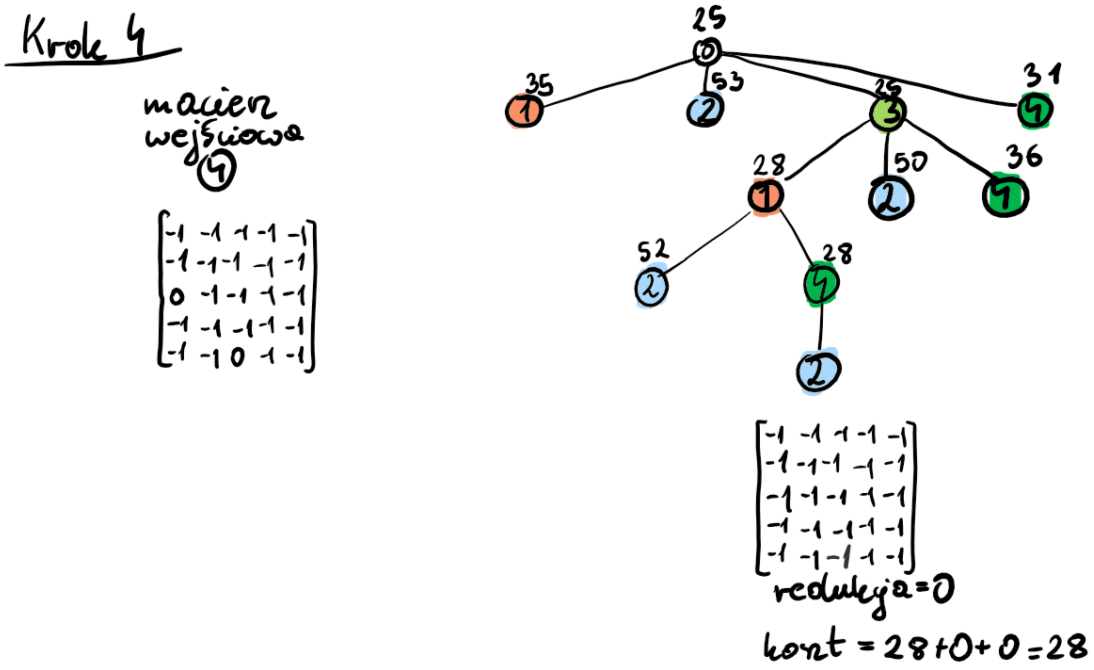
Rysunek 2. Krok drugi algorytmu – drugie rozwinięcie przestrzeni rozwiązań.



Rysunek 3. Krok trzeci algorytmu – trzecie rozwinięcie przestrzeni rozwiązań.

Trzeci krok algorytmu pokazany na rysunku 3, przedstawia dalsze rozwinięcie przestrzeni rozwiązań – tym razem dla węzła 1 jako rodzica i par węzłów (1-2, 1-4). Podobnie jak poprzednio obliczane są zredukowane macierze i koszty.

Rysunek 4 przedstawia końcowe rozwinięcie przestrzeni rozwiązań. Po dotarciu do ostatniego wierzchołka i utworzeniu pełnej ścieżki – algorytm zaktualizuje wartość górnej granicy, która wynosi w tym przypadku 28. Każdy inny koszt nieodwiedzonego węzła jest większy od 28, więc jest to rozwiązanie optymalne.



Rysunek 4. Krok czwarty algorytmu – ostatnie rozwinięcie przestrzeni rozwiązań.

Algorytm Programowania Dynamicznego

To podejście optymalizuje problem ATSP, wykorzystując rekurencję z zapamiętywaniem stanów, co zmniejsza liczbę powtarzanych obliczeń. Jego złożoność obliczeniowa wynosi $O(N^2 \cdot 2^n)$, co jest znaczną poprawą względem podejścia brute force, lecz nadal ma charakter wykładniczy, co ogranicza jego zastosowanie do średniej wielkości instancji. Zaimplementowany algorytm działa poprzez przeszukiwanie wszystkich możliwych ścieżek pomiędzy miastami, jednocześnie zapamiętując i minimalizując koszty podróży z określonym stanem odwiedzonych miast (reprezentowanym jako maska bitowa). Funkcja searchAllSolutions rekurencyjnie sprawdza każdą możliwą trasę z aktualnego miasta do innych jeszcze nieodwiedzonych, obliczając koszt podróży i oznaczając odwiedzone miasta poprzez modyfikację maski. Każdy stan (pozycja w mieście i zestaw odwiedzonych miast) jest zapisywany w tablicy cost_array, aby uniknąć ponownego obliczania tego samego przypadku. W trakcie działania algorytmu tablica path_array przechowuje informacje o kolejnych miastach na minimalnej trasie, co umożliwia późniejsze odtworzenie optymalnej ścieżki. Na końcu algorytm drukuje minimalny koszt oraz sekwencję odwiedzonych miast.

Kod 2. Implementacja algorytmu Programowania Dynamicznego.

```
void TSP::dynamicProgrammingTSP(cityMatrix &city_matrix) {
    int mask = 1;
    int city_position = 0;
    int min_path[city_matrix.size];

    // Initialize the cost and path arrays
    int** cost_array = new int*[city_matrix.size]; //n cities
    int** path_array = new int*[city_matrix.size]; //n cities
    for (int i = 0; i < (city_matrix.size); i++) {
        cost_array[i] = new int[1 << city_matrix.size]; //2^n masks
        path_array[i] = new int[1 << city_matrix.size]; //2^n masks
    }

    for (int row = 0; row < city_matrix.size; row++) { //initialize the
cost array
        for (int col = 0; col < 1 << city_matrix.size; col++) {
            cost_array[row][col] = -1;
            path_array[row][col] = -1;
        }
    }
    //-----------------------------------------

    int cost = searchAllSolutions(city_matrix, mask, city_position,
path_array, cost_array);
    std::cout << "Minimum cost: " << cost << std::endl;

    std::cout << "Path: ";
    printPath(city_matrix, path_array);

    for (int i = 0; i < city_matrix.size ; i++) {
        delete [] path_array[i];
        delete [] cost_array[i];
    }
    delete [] path_array;
    delete [] cost_array;

}
```

Badanie algorytmów

Plan eksperymentu

Badania przeprowadzone zostały dla siedmiu reprezentatywnych wielkości N dla tych samych danych (tj. każdy algorytm korzystał z tej samej macierzy w danej iteracji). Ze względu na znacznie dłuższy czas wykonania algorytmów Brute Force i Dynamicznego Programowania, niektóre z wartości N zostały dla nich pominięte. Dodatkowo, przeprowadzone zostały badania nad każdym z algorytmów z osobna, gdzie zostały dobrane inne reprezentatywne wartości N, bardziej dopasowane do każdego z algorytmów.

Reprezentatywne wartości N dla pierwszego badania: **5, 10, 12, 14, 18, 20, 22**

Reprezentatywne wartości N dla drugiego badania:
Brute Force: **5, 6, 7, 8, 9, 10, 11**
Dynamiczne Programowanie: **5, 10, 12, 15, 18, 20, 22**
Branch & Bound: **5, 10, 15, 20, 25, 28, 30**

Dla każdego algorytmu znaleziona zostanie maksymalna instancja problemu N, dla której algorytm wykonuje się w co najmniej 120 sekund.

Generowanie danych:

Struktury danych są generowane w funkcji createRandomMatrix(n), która tworzy macierz reprezentującą koszt przejścia pomiędzy miastami o rozmiarze n. Przed każdym uruchomieniem algorytmu tworzone są nowe macierze kosztów, aby wyniki były reprezentatywne dla różnych przypadków testowych.

Pomiar czasu:

Czas wykonywania algorytmu będzie mierzony dla każdej reprezentatywnej wartości N. Pomiar czasu rozpocznie się tuż przed uruchomieniem samego algorytmu i zakończy się po zakończeniu obliczeń z nim związanych, bez uwzględnienia czasu generacji macierzy ani czasu kompilacji. Do pomiaru czasu użyjemy wbudowanego narzędzia standardowej biblioteki - std::chrono::high_resolution_clock.

Uśrednianie czasów wykonania:

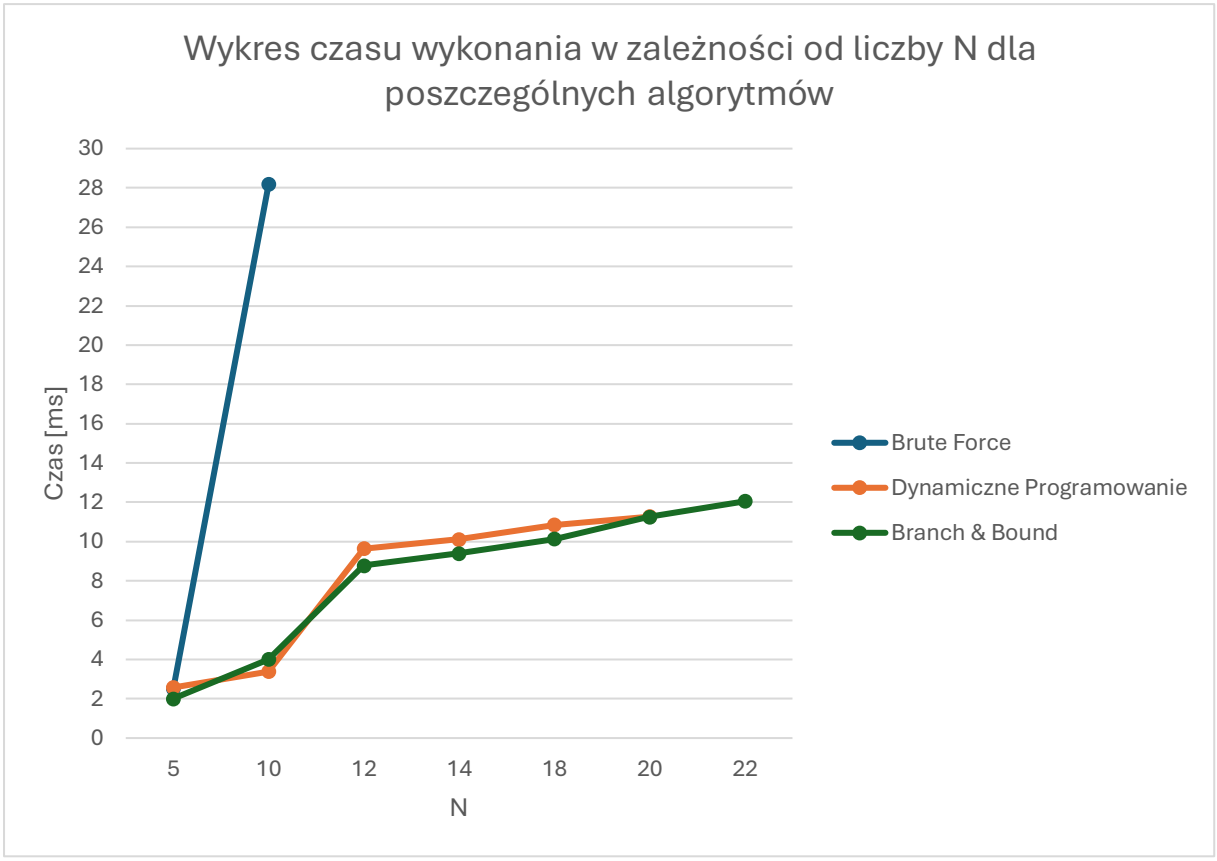
Badania czasów wykonania zostaną powtórzone stukrotnie dla każdego z rozmiarów N w badaniu pierwszym oraz pięćdziesięciokrotnie w badaniu drugim aby zminimalizować wpływ czynników losowych na wyniki.

Wyniki badań

Tabela 1 zawiera wyniki badania pierwszego. Wyniki dla algorytmu Brute Force pomijają wartości N od 14 w górę z uwagi na bardzo długi czas wykonania. W dodatku na wykresie 1 pominięty został również czas dla N = 12 w celu zwiększenia jego czytelności.

Tabela 1. Uśrednione czasy wykonania algorytmów ATSP w zależności od rozmiaru problemu N.

N	Brute Force [ms]	Dynamiczne Programowanie [ms]	Branch & Bound [ms]
5	2.52448	2.57416	2.00841
10	28.187	3.39042	4.00129
12	4115.66	9.64351	8.77683
14	-	10.11	9.39811
18	-	10.8418	10.1358
20	-	11.268	11.2547
22	-	-	12.0487

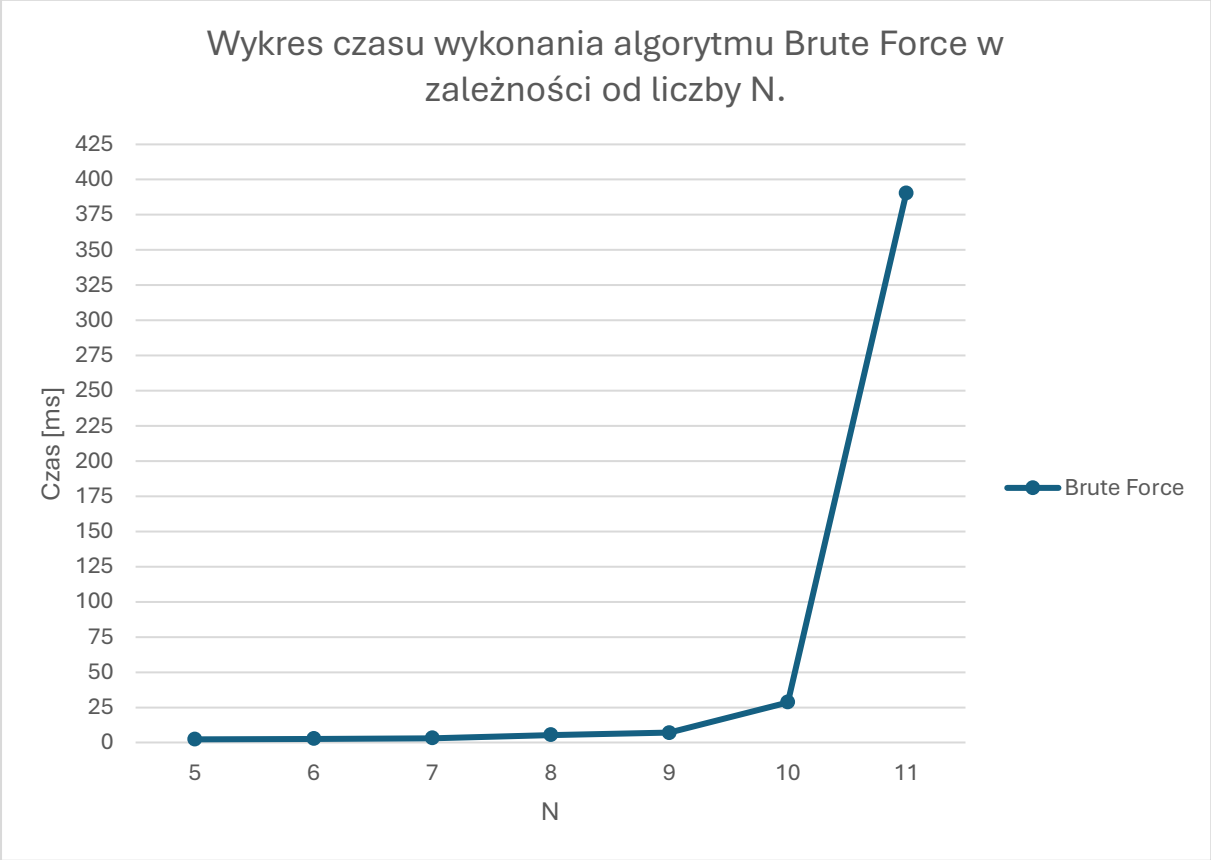


Wykres 1. Czas wykonania algorytmów ATSP w zależności od rozmiaru problemu N.

Tabele 2, 3 oraz 4 zawierają wyniki badania drugiego. Każda z nich przedstawiona została na wykresach 2 do 4.

Tabela 2. Uśrednione czasy wykonania algorytmu Brute Force w zależności od rozmiaru problemu N.

Brute Force							
N	5	6	7	8	9	10	11
Czas [ms]	2.24008	2.66776	3.0827	5.40323	6.90703	28.6025	390.209



Wykres 2. Czas wykonania algorytmu Brute Force w zależności od rozmiaru problemu N.

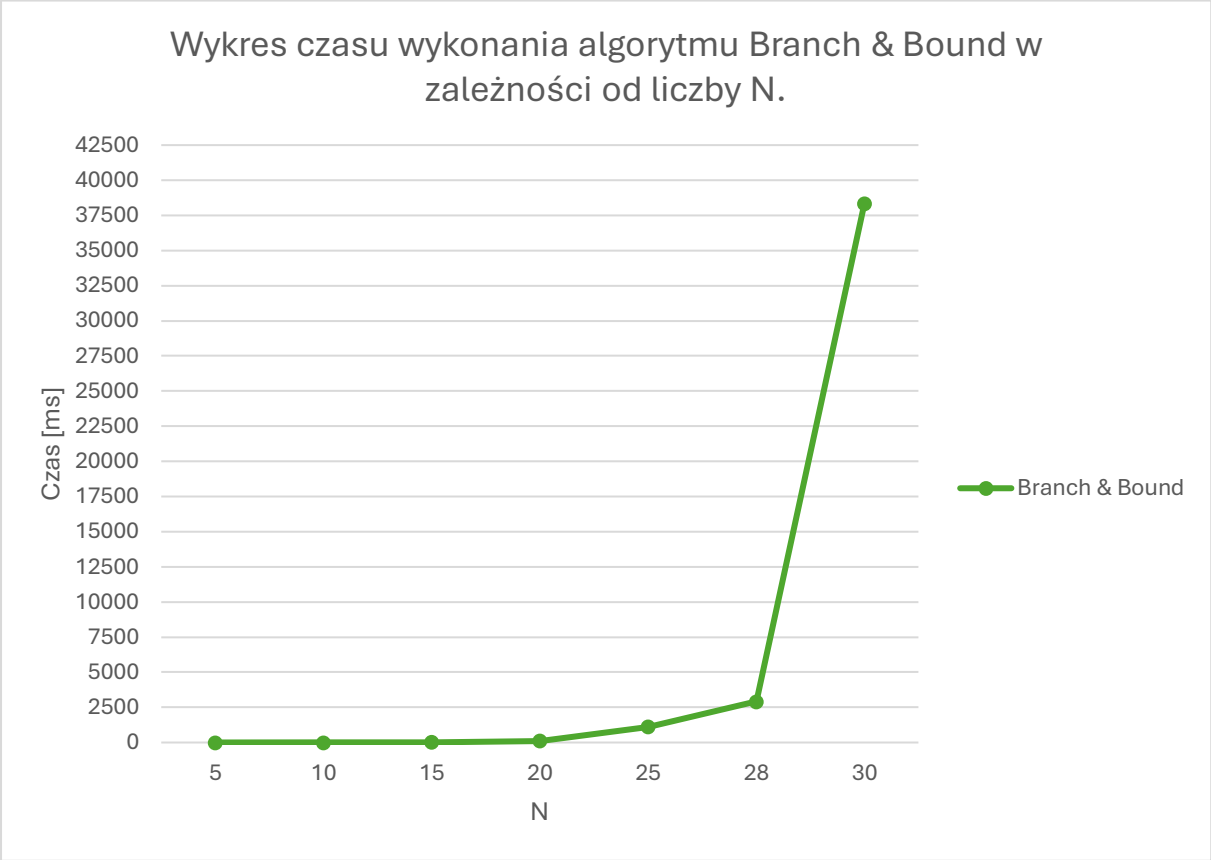
Tabela 3. Uśrednione czasy wykonania algorytmu Programowania Dynamicznego w zależności od rozmiaru problemu N.

Programowanie Dynamiczne							
N	5	10	12	15	18	20	22
Czas [ms]	5.08797	7.77619	11.0409	56.8936	595.378	4645.25	13015.8

Wykres 3. Czas wykonania algorytmu Programowania Dynamicznego w zależności od rozmiaru problemu N.

Tabela 4. Uśrednione czasy wykonania algorytmu Branch & Bound w zależności od rozmiaru problemu N.

Branch & Bound							
N	5	10	15	20	25	28	30
Czas [ms]	2.27732	4.43812	15.8683	114.147	1106.86	2912.39	38319.4



Wykres 4. Czas wykonania algorytmu Branch & Bound w zależności od rozmiaru problemu N.

W tabeli 5 znajdują się wyniki wyznaczenia maksymalnego N dla którego każdy z poszczególnych algorytmów wykonuje się w czasie co najmniej 120s. Uwaga, w tabeli występują czasy dla przykładowego wywołania.

Tabela 5. Maksymalna liczba N, dla którego poszczególne algorytmy wykonują się w określonym czasie (120s)

	Max N	Czas [s]
Branch & Bound	13	24.4785
Programowanie Dynamiczne	25	101.437
Brute Force	37	103.163

Podsumowanie

Algorytm siłowy (Brute Force):

- Algorytm Brute Force wykazuje znacznie dłuższe czasy wykonania już przy stosunkowo niewielkich wartościach N. Wyniki z Tabeli 2 wskazują, że przy N=11 czas wykonania wynosi już ponad 390 ms, co przy dalszym wzroście N rośnie w sposób wykładniczy.
- Maksymalna liczba miast, dla której algorytm Brute Force wykonuje się w czasie poniżej 120 sekund, wynosi N=13 (zgodnie z Tabelą 5). Po osiągnięciu tego progu, algorytm staje się niepraktyczny ze względu na czasochłonność.

Programowanie dynamiczne:

- Algorytm programowania dynamicznego jest bardziej efektywny niż Brute Force, co widać w porównaniu czasów dla większych wartości N. W Tabeli 1 dla N=12 jego czas wykonania wynosi około 9,64 ms, podczas gdy dla Brute Force to ponad 4115 ms.
- Wyniki w Tabeli 5 pokazują, że algorytm ten radzi sobie dobrze do N=25, po czym czasy wykonania przekraczają próg 120 sekund.

Algorytm Branch & Bound:

- W porównaniu do poprzednich metod, algorytm Branch & Bound wypada najlepiej pod względem skalowalności przy rosnącym N. Tabela 4 pokazuje, że czasy wykonania zaczynają rosnąć szybciej dopiero przy większych wartościach N. Dla N=20 czas to około 114,1 ms, a dla N=25 już 1106,86 ms.
- Z Tabeli 5 wynika, że algorytm Branch & Bound jest efektywny do N=36, co czyni go najlepszym wyborem przy dużych instancjach, dopóki jego czasy wykonania mieszczą się w akceptowalnych granicach.

Badanie potwierdziło, że efektywność algorytmów rozwiązujących problem komiwojażera drastycznie różni się w zależności od rozmiaru instancji. Algorytm Brute Force sprawdza się tylko dla małych wartości N, programowanie dynamiczne jest wydajniejsze dla większych problemów, a Branch & Bound zapewnia największe możliwości, jeżeli chodzi o rozwiązywanie problemów o dużym parametrze N.

Bibliografia

- https://www.youtube.com/watch?v=1FEP_sNb62k&ab_channel=AbdulBari
- https://www.youtube.com/watch?v=XaXsJJh-Q5Y&t=6s&ab_channel=AbdulBari
- https://www.youtube.com/watch?v=JE0JE8ce1V0&t=1515s&ab_channel=CodingBlocks
- https://www.youtube.com/watch?v=-cLsEHP0qt0&ab_channel=nptelhrd
- https://www.youtube.com/watch?v=nN4K8xA8ShM&ab_channel=nptelhrd
- https://dspace.mit.edu/bitstream/handle/1721.1/46828/algorithmfortrav00litt.pdf?sequence=1&isAllowed=y
- https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/
- https://cs.pwr.edu.pl/zielinski/lectures/om/mow10.pdf