Politechnika Wrocławska

Sprawozdanie z Projektu 2

Przedmiotu Projektowanie Efektywnych Algorytmów

Temat: Rozwiązanie asymetrycznego problemu komiwojażera metodą tabu search.

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

Kyrylo Semenchenko 273004
Informatyka Techniczna I stopnia
Wydział Informatyki i Telekomunikacji

11 grudnia 2024

Spis treści

1 Wstęp Teoretyczny **3**
1.1 Opis Rozpatrywanego Problemu . . . 3
1.2 Opis Algorytmu Tabu Search . . . 3
1.3 Oszacowanie Złożoności Obliczeniowej . . . 4

2 Implementacja Algorytmu **5**
2.1 Klasa TabuSearch . . . 5
2.1.1 Podstawowe pola klasy . . . 5
2.1.2 Kluczowe metody . . . 6
2.2 Klasa Matrix . . . 6
2.2.1 Podstawowe pola klasy . . . 6
2.2.2 Kluczowe metody . . . 6
2.2.3 Losowa generacja macierzy . . . 7
2.2.4 Wczytywanie danych z pliku . . . 7
2.3 Struktury danych . . . 7

3 Przykład Praktyczny **7**
3.1 Tabu Search . . . 7
3.1.1 1. Inicjalizacja . . . 7
3.1.2 2. Iteracja 1 . . . 8
3.1.3 3. Iteracja 2 . . . 8

4 Wyniki Eksperymentów **8**
4.1 Specyfikacja sprzętowa 8

5 Eksperymenty i Analiza Wyników **9**
5.1 Metodyka Eksperymentu 9
5.2 Tabela wyników 9
5.3 Wykresy Wyników 9
5.4 Analiza Wyników 11

6 Wnioski **11**
6.1 Wydajność Algorytmów 11
6.2 Zastosowania Praktyczne 11
6.3 Wnioski Końcowe 11

7 Literatura **12**

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 Tabu Search

Algorytm *Tabu Search* jest metaheurystyką stosowaną do rozwiązywania problemów optymalizacyjnych, takich jak problem asymetrycznego komiwojażera (ATSP). Celem algorytmu jest znalezienie rozwiązania optymalnego, unikając lokalnych minimów i umożliwiając eksplorację przestrzeni rozwiązań.

Tabu Search działa iteracyjnie, eksplorując przestrzeń rozwiązań w sąsiedztwie bieżącego rozwiązania. W algorytmie wykorzystuje się *listę tabu*, która przechowuje pewne ruchy (zamiany miast w ścieżce), aby uniknąć cyklicznego przeszukiwania tych samych rozwiązań.

Zastosowanie *kryterium aspiracji* pozwala na zignorowanie ruchu tabu, jeśli prowadzi on do lepszego rozwiązania niż dotychczasowe najlepsze.

Główne kroki algorytmu Tabu Search:

1. **Inicjalizacja**:
   - Wygenerowanie losowego rozwiązania początkowego (ścieżka odwiedzanych miast, zaczynająca i kończąca się w mieście startowym).
   - Ustawienie parametrów algorytmu: długości listy tabu (*tabuListSize*) oraz maksymalnej liczby iteracji (*maxIterations*).
   - Zainicjalizowanie listy tabu jako pustej.
2. **Przeszukiwanie sąsiedztwa**:
   - W każdej iteracji algorytm przeszukuje sąsiedztwo bieżącego rozwiązania, generując ruchy polegające na zamianie dwóch miast w ścieżce.
   - Ruchy tabu są odrzucane, chyba że spełniają kryterium aspiracji.
   - Z sąsiedztwa wybierany jest ruch prowadzący do najlepszego rozwiązania lokalnego.
3. **Aktualizacja rozwiązania**:
   - Najlepszy ruch zostaje zastosowany do bieżącej ścieżki.
   - Ruch zostaje dodany do listy tabu, a najstarszy wpis jest usuwany (FIFO).
   - Jeśli koszt bieżącej ścieżki jest lepszy niż dotychczasowy najlepszy wynik, rozwiązanie globalne zostaje zaktualizowane.
4. **Kryterium stopu**: Algorytm kończy działanie, gdy osiągnięta zostanie maksymalna liczba iteracji lub inne warunki zakończenia, np. brak poprawy rozwiązania przez określoną liczbę iteracji.

Dodatkowe elementy algorytmu:

- Kryterium *aspiracji* pozwala na wykonanie ruchu tabu, jeśli prowadzi on do rozwiązania o lepszym koszcie niż dotychczas najlepsze.
- Losowe przeszukiwanie sąsiedztwa w celu zmniejszenia złożoności czasowej, poprzez próbkowanie ograniczonej liczby ruchów.

Algorytm działa w pętli, poszukując lepszych rozwiązań w przestrzeni przeszukiwania, a jego efektywność zależy od właściwego doboru parametrów, takich jak długość listy tabu oraz liczba iteracji.

1.3 Oszacowanie Złożoności Obliczeniowej

Złożoność obliczeniowa algorytmu Tabu Search zależy od liczby iteracji oraz wielkości przestrzeni sąsiedztwa przeszukiwanej w każdej iteracji.

- **Generowanie sąsiedztwa**: Dla problemu ATSP, sąsiedztwo definiowane jest poprzez ruchy polegające na zamianie dwóch miast w ścieżce. Maksymalna liczba takich zamian wynosi:

$$\mathcal{O}(n^2)$$

gdzie n to liczba miast.

- **Ograniczone przeszukiwanie sąsiedztwa**: W implementacji zastosowano losowe próbkowanie sąsiedztwa. Liczba ruchów do oceny została ograniczona do $k = \frac{n(n-1)}{3}$, co zmniejsza efektywną liczbę analizowanych ruchów.

- **Ocena rozwiązania**: Koszt każdej ścieżki obliczany jest w czasie $\mathcal{O}(n)$, ponieważ ścieżka składa się z n przejść między miastami.

- **Całkowita złożoność**: Dla $maxIterations$ iteracji i ograniczonej liczby ruchów (k), całkowita złożoność wynosi:

$$\mathcal{O}(maxIterations \cdot k \cdot n)$$

Ponieważ $k \sim \mathcal{O}(n^2)$ (przybliżona liczba ruchów), ostateczna złożoność to:

$$\mathcal{O}(maxIterations \cdot n^3)$$

W praktyce, ograniczenie liczby analizowanych ruchów oraz zastosowanie listy tabu pozwala na znaczącą redukcję czasu działania algorytmu w porównaniu z pełnym przeszukiwaniem sąsiedztwa.

2 Implementacja Algorytmu

2.1 Klasa TabuSearch

Klasa `TabuSearch` implementuje algorytm Tabu Search w celu rozwiązania asymetrycznego problemu komiwojażera (ATSP).

2.1.1 Podstawowe pola klasy

- `const Matrix& matrix` – Referencja do macierzy kosztów reprezentującej problem.
- `int size` – Liczba miast w problemie (rozmiar macierzy).
- `int bestCost` – Najlepszy koszt znaleziony podczas przeszukiwania.
- `int* bestPath` – Najlepsza ścieżka znaleziona podczas przeszukiwania.
- `int* currentPath` – Aktualnie oceniana ścieżka.
- `int tabuListSize` – Rozmiar listy tabu.
- `int** tabuList` – Lista tabu przechowująca niedozwolone ruchy.
- `int maxIterations` – Maksymalna liczba iteracji algorytmu.

2.1.2 Kluczowe metody

- `TabuSearch(const Matrix& matrix, int tabuListSize, int maxIterations)` – Konstruktor inicjalizujący obiekt algorytmu.
- `const int* runTabuSearch()` – Uruchamia algorytm Tabu Search i zwraca najlepsze rozwiązanie.
- `void printSolutionTabu() const` – Wypisuje najlepsze znalezione rozwiązanie.
- `bool isTabu(int i, int j)` – Sprawdza, czy ruch jest zabroniony (znajduje się na liście tabu).
- `void updateTabuList(int i, int j)` – Aktualizuje listę tabu po wykonaniu ruchu.
- `int calculateCost(const int* path)` – Oblicza koszt danej ścieżki.
- `void swapCities(int* path, int i, int j)` – Wymienia dwa miasta w aktualnej ścieżce.

2.2 Klasa Matrix

Klasa `Matrix` odpowiada za przechowywanie danych wejściowych algorytmu w postaci macierzy kosztów.

2.2.1 Podstawowe pola klasy

- `int size` – Rozmiar macierzy, odpowiadający liczbie miast.
- `int* data` – Wskaźnik na 1-wymiarową tablicę przechowującą dane macierzy.

2.2.2 Kluczowe metody

- `Matrix(int s)` – Konstruktor, który alokuje pamięć dla macierzy o rozmiarze `s`.
- `void readFromFile(const std::string& filename)` – Wczytuje macierz kosztów z pliku.
- `void generateRandomMatrix(int s, int minValue, int maxValue, int symmetricity, int asymRangeMin, int asymRangeMax)` – Generuje losową macierz o zadanym rozmiarze i właściwościach.
- `int getCost(int i, int j) const` – Zwraca koszt przejścia z miasta `i` do miasta `j`.
- `void display() const` – Wyświetla macierz kosztów w czytelnej formie.

2.2.3 Losowa generacja macierzy

Losowa generacja macierzy pozwala na kontrolowanie stopnia symetryczności oraz zakresu asymetryczności kosztów. Funkcja wykorzystuje:

- `std::uniform_int_distribution` – Losowe generowanie wartości kosztów.
- `std::mt19937` – Generator liczb pseudolosowych.

2.2.4 Wczytywanie danych z pliku

Metoda `readFromFile` pozwala wczytać dane problemu z pliku CSV. Format pliku zakłada pierwszą linię z rozmiarem macierzy, a następnie wiersze reprezentujące wiersze macierzy.

2.3 Struktury danych

Dane algorytmu przechowywane są w następujący sposób:

- **Macierz kosztów** – Jednowymiarowa tablica liczb całkowitych symulująca dwuwymiarową macierz.
- **Lista tabu** – Tablica dwuwymiarowa przechowująca ruchy niedozwolone w kolejnych iteracjach.
- **Ścieżki** – Tablice dynamiczne, które przechowują aktualną i najlepszą znalezioną trasę.

3 Przykład Praktyczny

3.1 Tabu Search

Rozważmy problem ATSP (Asymmetric Traveling Salesman Problem) dla poniższej macierzy kosztów:

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

Legenda:

- $D[i][j]$: koszt przejścia z miasta i do miasta j.
- -1: brak możliwości przejścia (miasta nie odwiedzają się same).

3.1.1 1. Inicjalizacja

- Początkowa ścieżka: $[0, 1, 2, 3]$ (miasta numerujemy od 0).
- Koszt początkowy:

$$C(0, 1, 2, 3) = D[0][1] + D[1][2] + D[2][3] + D[3][0] = 10 + 9 + 12 + 8 = 39$$

- Lista tabu: Pusta.
- Najlepsze rozwiązanie (początkowo): $([0, 1, 2, 3], 39)$.

3.1.2 2. Iteracja 1

Generowanie sąsiedztwa Sąsiedztwo generujemy poprzez zamianę dwóch miast w bieżącej ścieżce $[0, 1, 2, 3]$:

- $swap(1, 2) \rightarrow [0, 2, 1, 3]$:

$$C(0, 2, 1, 3) = D[0][2] + D[2][1] + D[1][3] + D[3][0] = 15 + 13 + 10 + 8 = 46$$

- $swap(1, 3) \rightarrow [0, 3, 2, 1]$:

$$C(0, 3, 2, 1) = D[0][3] + D[3][2] + D[2][1] + D[1][0] = 20 + 9 + 13 + 5 = 47$$

- $swap(2, 3) \rightarrow [0, 1, 3, 2]$:

$$C(0, 1, 3, 2) = D[0][1] + D[1][3] + D[3][2] + D[2][0] = 10 + 10 + 9 + 6 = 35$$

Wybór najlepszego sąsiada Spośród sąsiadów wybieramy rozwiązanie o najniższym koszcie, które nie znajduje się na liście tabu:

- Najlepszy sąsiad: $[0, 1, 3, 2]$ o koszcie 35.

Aktualizujemy rozwiązanie bieżące i dodajemy ruch $swap(2, 3)$ do listy tabu.

3.1.3 3. Iteracja 2

Powtarzamy proces generowania sąsiedztwa dla nowego rozwiązania bieżącego $[0, 1, 3, 2]$, pamiętając o ograniczeniach listy tabu.

Generowanie sąsiedztwa Dalsze kroki są analogiczne do powyższych, a proces iteracji trwa do osiągnięcia warunku stopu, takiego jak liczba iteracji lub brak poprawy rozwiązania przez określoną liczbę kroków.

4 Wyniki Eksperymentów

4.1 Specyfikacja sprzętowa

Wszystkie pomiary algorytmów wykonane w języku C++ na laptopie z następującymi parametrami:

- Procesor: Intel Core i5-12450 (bazowa częstotliwość 2.00 GHz)
- Pamięć RAM: 16 GB
- System operacyjny: Windows 10
- Środowisko programistyczne: Visual Studio Code
- Budowanie: cmake 3.29.2
- Kompilator: g++ 13.2.0
- Typ budowania: Release

Obliczenia, wykresy oraz analizę wyników przeprowadzono w języku Python 3.12.4 z wykorzystaniem bibliotek: `pandas, numpy, seaborn.`

5 Eksperymenty i Analiza Wyników

Dla eksperymentów zostały użyte macierze sąsiedztwa dostępne na stronie *TspLib - Universität Heidelberg*. W ramach badań wykorzystano osiem macierzy różnych rozmiarów, od 17 do 71.

Na podstawie artykułu *A Tabu Search Algorithm for Cluster Building in Wireless Sensor Networks (str. 5–6)* stwierdzono, że dostosowanie rozmiaru kolejki `tabuList` indywidualnie do każdego problemu na poziomie około 75% rozmiaru problemu pozytywnie wpłynęło na dokładność uzyskanych najlepszych ścieżek. Jednakże zwiększenie rozmiaru kolejki skutkowało proporcjonalnym wzrostem czasu wykonania algorytmu.

Macierz kosztów przechowuje wartości w pamięci w postaci tabeli dwuwymiarowej, co zwiększyło wydajność przeszukiwania w porównaniu do wcześniej używanych wektorów. Wszystkie algorytmy oraz struktury zostały zaimplementowane w języku C++ bez użycia dodatkowych bibliotek zewnętrznych.

5.1 Metodyka Eksperymentu

Dla eksperymentów zostały użyte macierze sąsiedztwa przedstawione na stronie *TspLib - Universität Heidelberg*. W ramach badań wykorzystano 8 macierzy o różnych rozmiarach, od 17 do 71. Na podstawie artykułu *A Tabu Search Algorithm for Cluster Building in Wireless Sensor Networks (str. 5–6)*, zastosowano dopasowanie rozmiaru kolejki `tabuList` do każdego problemu indywidualnie na poziomie ok. 75% rozmiaru problemu. Podejście to pozytywnie wpłynęło na dokładność uzyskanych najlepszych ścieżek, ale spowodowało wzrost czasu wykonania algorytmu proporcjonalny do zwiększenia rozmiaru problemu.

Macierz kosztów była przechowywana w pamięci w postaci tabeli dwuwymiarowej, co zwiększyło wydajność przeszukiwania w porównaniu do wcześniej używanych wektorów. Wszystkie algorytmy oraz struktury zostały zaimplementowane bez dodatkowych bibliotek zewnętrznych w języku C++.

W ramach eksperymentu mierzono wartość najlepszej ścieżki uzyskanej podczas działania algorytmu oraz czas wykonania algorytmu. Do pomiaru czasów wykonania użyto biblioteki `chrono` z języka C++, która umożliwia dokładne mierzenie czasu wykonania kodu.

Dla każdego problemu wykonano 50 próbek, a wyniki końcowe zostały uśrednione z użyciem skryptów napisanych w Pythonie. Obliczono średni błąd bezwzględny oraz względny dla rozmiaru ścieżki uzyskanej dla każdego problemu. Wyniki zapisano w pliku `csv`, a wykresy wygenerowano również z użyciem Pythona.

5.2 Tabela wyników

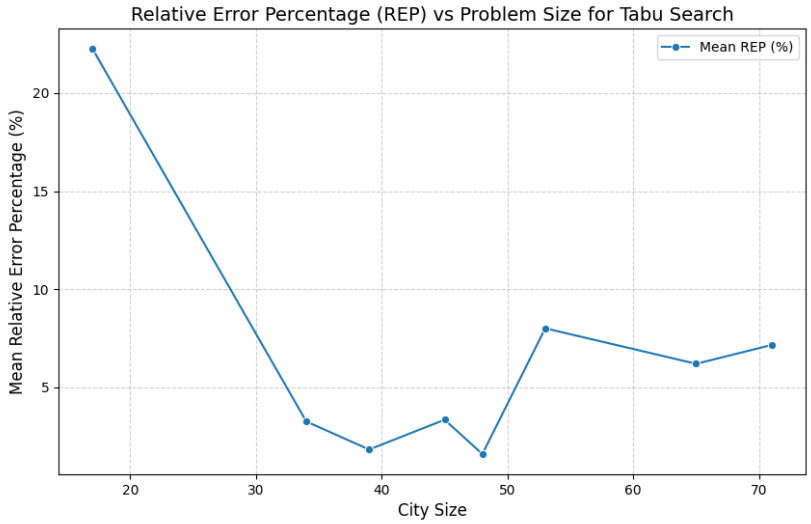
Dane wyników algorytmu Tabu Search zostały zapisane w pliku `tabuResultsTable.csv`. Poniżej przedstawiono dane w formie tabeli (Tabela 1).

5.3 Wykresy Wyników

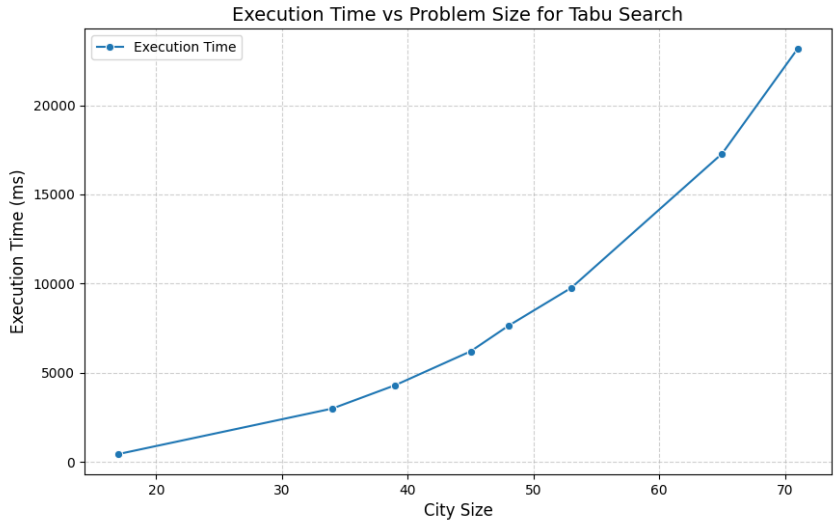
W tej sekcji przedstawiono dwa wykresy ilustrujące wyniki algorytmu Tabu Search.

Rozmiar miast	Prawidłowa odpowiedź	Śr. najlepsza ścieżka	Śr. błąd bezwzględny	Śr. błąd względny (%)	Śr. czas (ms)
17	39	47.68	8.68	22.26	445.24
34	1286	1327.8	41.8	3.26	2995.26
39	1530	1557.86	27.86	1.83	4304.53
45	1613	1667.02	54.02	3.35	6204.29
48	14422	14652.56	230.56	1.6	7633.27
53	6905	7457.95	552.95	8.01	9762.23
65	1839	1952.84	113.84	6.2	17292.08
71	1950	2089.48	139.49	7.16	23168.42

Tabela 1: Wyniki działania algorytmu Tabu Search dla różnych rozmiarów miast.



Rysunek 1: Relatywny błąd procentowy (REP) w zależności od rozmiaru problemu dla algorytmu Tabu Search.



Rysunek 2: Czas wykonania w zależności od rozmiaru problemu dla algorytmu Tabu Search.

5.4 Analiza Wyników

Na podstawie uzyskanych wyników, przedstawionych w tabeli 1 oraz na wykresach 1 i 2, można wywnioskować, że algorytm *Tabu Search* nie jest szczególnie dokładny dla mniejszych instancji. Jednak wraz ze wzrostem rozmiaru problemu zmniejsza się błąd względny. Ten spadek świadczy o skuteczności metaheurystycznego podejścia w budowaniu algorytmów dla dużych instancji problemów.

Warto zauważyć, że czas wykonania algorytmu rośnie w sposób zbliżony do liniowego wraz z rozmiarem problemu. Jest to spowodowane przyjętą metodologią symulacji, w szczególności dynamicznym rozmiarem kolejki `tabuList`.

6 Wnioski

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

6.1 Wydajność Algorytmów

Algorytm `Tabu Search` okazał się szczególnie skuteczny w przypadku większych instancji problemu. Pomimo wyższych czasów wykonania, zastosowanie dynamicznie dobieranej kolejki `tabuList` poprawiło jakość uzyskanych rozwiązań, zmniejszając błąd względny wraz ze wzrostem rozmiaru problemu. Wyniki wskazują na quasi-liniowy wzrost czasu wykonania algorytmu w zależności od wielkości instancji, co jest zgodne z oczekiwaniami dla podejścia pseudoheurystycznego.

6.2 Zastosowania Praktyczne

Algorytm `Tabu Search` może być stosowany w praktycznych problemach optymalizacyjnych, takich jak logistyka, planowanie tras czy optymalizacja sieci. Jego elastyczność w dostosowywaniu parametrów pozwala na aplikację w różnych środowiskach problemowych. Warto jednak zauważyć, że dla bardzo dużych instancji czas wykonania może stać się ograniczeniem, dlatego konieczne jest dalsze doskonalenie implementacji oraz optymalizacja algorytmu pod kątem czasu działania.

6.3 Wnioski Końcowe

Algorytm `Tabu Search` udowodnił swoją skuteczność w rozwiązywaniu problemów takich jak ATSP, szczególnie dla dużych instancji. Dobrze dostosowane parametry, takie jak długość kolejki `tabuList`, mają kluczowe znaczenie dla osiągnięcia wysokiej jakości wyników. Metoda ta jest wartościowym narzędziem w rozwiązywaniu złożonych problemów optymalizacyjnych. W przyszłości warto zbadać możliwości hybrydyzacji algorytmu z innymi metodami heurystycznymi oraz wykorzystanie bardziej zaawansowanych struktur danych w celu dalszej poprawy wydajności. Należy również rozważyć inne algorytmy przeszukiwania lokalnego, które prszeszukwują przestrzeń rozwiązań w inny sposób niż `swap`

7 Literatura

1 Agung Chandra and Christine Natalia, *The Study of Depot Position Effect on Travel Distance in Order Picking Problem*

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

3 Abdelmorhit El Rhazi and Samuel Pierre, *A Tabu Search Algorithm for Cluster Building in Wireless Sensor Networks*, Senior Member, IEEE. (str 5-6)

4 Paweł Zieliński, *Metody przeszukiwania lokalnego.*

5 *TspLib - Universität Heidelberg*