Projekt z Projektowania Efektywnych Algorytmów

Numer etapu: 2

Zadanie wykonał: Michał Wróbel (259132)

Grupa laboratoryjna: K00-490

Termin zajęć: WT 17.05

Data oddania: 13.12.2022

Prowadzący: Dr inż. Jarosław Mierzwa

1. **Cel:**

   Celem projektu było zaimplementowanie algorytmu tabu z dywersyfikacją i trzema sąsiedztwami dla asymetrycznego problemu komiwojażera. Algorytm został zaimplementowany w języku Python.

2. **Projekt:**

   a) **Wstęp teoretyczny**

   Metaheurystyka Tabu Search może być określona jako metoda dynamicznej zmiany sąsiedztwa danego rozwiązania. Oznacza to, że zbiór N(x) nie jest z góry ustalony dla każdego x lecz może zmieniać się w zależności od informacji zebranych w poprzednich etapach przeszukiwania.

   Algorytm ten posiada między innymi listę tabu, czyli listę zakazanych ruchów. Ruchy umieszczone na tej liście przez czas kadencji nie mogą być ponownie użyte przy wyznaczaniu nowej ścieżki. W mojej implementacji długość kadencji to długość macierzy podzielona przez 4 w myśl zasady, że im większy rozmiar problemu tym kadencja powinna być dłuższa.

   Zostało również ustanowione kryterium aspiracji. Wg. niego sąsiednie rozwiązanie może zostać zaakceptowane, gdy jest najlepszym rozwiązaniem znalezionym w dotychczasowym cyklu przeszukiwań, lub jeżeli znajduje się na liście tabu, ale prowadzi do wyniku lepszego niż jakikolwiek dotychczas rozpatrywane rozwiązanie.

   Strategia dywersyfikacji w moim algorytmie polega na tym, że jeżeli ilość iteracji bez polepszenia rozwiązania wynosi zadaną wartość (jest to 3-krotnośc długości macierzy), to generujemy nowe startowe rozwiązanie metodą zachłanną (pierwsze miasto jest dobierane w sposób losowy). Strategia ta jest potrzebna na wypadek

tego, gdybyśmy się znaleźli w obrębie minimum lokalnego, aby dalej kontynuować przeszukiwania.

Sąsiednie rozwiązania są w przypadku tej implementacji za każdym razem generowane w sposób losowy (losujemy dwa miasta). Następnie w zależności od tego w której opcji pracuje algorytm albo dokonuje w aktualnie przeglądanej ścieżce zamiany ze sobą dwóch miast (swap), albo odwraca kolejność miast pomiędzy tymi dwoma miastami (invert), albo wstawia jedno miasto przed drugie (insert).

Tutaj najważniejsze fragmenty kod w raz z objaśnieniami w komentarzach:

```python
@staticmethod
# Algorytm wyznaczający metodą zachłanną startową ścieżkę
def start_path(matrix):

    # Miasto startowe.
    previous_city = random.randint(0, len(matrix) - 1)

    # Miasta, których nie odwiedziliśmy.
    remaining_cities = set(range(len(matrix)))

    # Aktualna ścieżka.
    path = []

    # Dopóki nie odwiedzimy wszystkich miast, wykonujemy następujące akcje.
    while len(remaining_cities) > 0:

        # Najlepsze miasto.
        best_city = []

        # Koszt podróży z poprzedniego do najlepszego miasta
        best_city_cost = float('inf')

        # Dla każdego miasta wśród pozostałych miast wykonujemy następujące akcje.
        for next_city in remaining_cities:

            # Jeżeli koszt podróży do następnego miasta jest mniejszy od aktualnie najlepszego kosztu
            if matrix[previous_city][next_city] < best_city_cost:
                best_city_cost = matrix[previous_city][next_city]
                best_city = next_city

        # Do ścieżki wstawiamy miasto, do którego koszt podróż jest najmniejszy
        path.append(best_city)

        # Jako poprzednie miasto ustawiamy najlepsze miasto.
        previous_city = best_city

        # Od listy pozostałych do rozpatrzenia miast odejmujemy najlepsze miasto
        remaining_cities = remaining_cities - {best_city}

    # Zwracamy ścieżkę.
    return path
```

```python
# Funkcja zamieniająca miejscami wybrane miasta
@staticmethod
def swap_neighbours(neighbour, path):
    # Wyciągnięcie informacji na temat tego, które wierzchołki mamy.
    first_city = int(neighbour[0])
    second_city = int(neighbour[1])

    # Pobranie aktualnej ścieżki i dokonanie w niej zmiany na podstawie informacji sąsiada.
    neighbour_path = path.copy()
    neighbour_path[first_city], neighbour_path[second_city] = neighbour_path[second_city], neighbour_path[first_city]

    return neighbour_path

# Funkcja wstawiająca wybrane miasto w daną pozycję
@staticmethod
def insert_neighbours(neighbour, path):
    # Wyciągnięcie informacji na temat tego, które wierzchołki mamy.
    city_position = int(neighbour[0])
    new_position = int(neighbour[1])

    # Pobranie aktualnej ścieżki i dokonanie w niej zmiany na podstawie informacji sąsiada.
    neighbour_path = path.copy()

    # Zapisujemy numer miasta, który przenosimy.
    city = neighbour_path[city_position]

    # Usuwamy to miasto z listy.
    neighbour_path.pop(city_position)

    # Wstawiamy miasto na nową pozycję.
    neighbour_path.insert(new_position, city)

    return neighbour_path
```

```python
# Funkcja odwracająca zadany fragment ścieżki
@staticmethod
def invert_neighbours(neighbour, path):
    # Wyciągnięcie informacji na temat tego, które wierzchołki mamy.
    start_position = int(neighbour[0])
    end_position = int(neighbour[1])

    # Zamiana miejsce, jeżeli startowa pozycja jest mniejsza od końcowej
    if start_position > end_position:
        start_position, end_position = end_position, start_position

    # Pobranie aktualnej ścieżki i dokonanie w niej zmiany na podstawie informacji sąsiada.
    neighbour_path = path.copy()
    if start_position == 0:
        neighbour_path = list(reversed(neighbour_path[start_position:end_position])) + neighbour_path[end_position:]
    elif end_position == (len(path) - 1):
        neighbour_path = neighbour_path[0:start_position] + neighbour_path[end_position:(start_position - 1):-1]
    elif start_position == 0 and end_position == (len(path) - 1):
        neighbour_path = neighbour_path[end_position:(start_position - 1):-1]
    else:
        neighbour_path = neighbour_path[0:start_position] + neighbour_path[end_position:(start_position - 1):-1] + neighbour_path[(end_position + 1):]
    return neighbour_path
```

```python
# Algorytm tabu search.
def tabu_search_algorithm(self, matrix, maximal_time, neighbour_type):

    # Maksymalna długość tabu list (i tym samym kadencji zakazanego rozwiązania).
    cadence_length = round(len(matrix) / 4)

    # Licznik iteracji bez poprawy wyniku.
    iteration_without_improvement_counter = 0

    # Po ilu iteracjach bez poprawy należy wygenerować nowe rozwiązanie
    max_iteration_without_improvement = len(matrix) * 3

    # Pierwsze wyznaczenie aktualnej ścieżki
    actual_path = self.start_path(matrix)

    # Pierwsze wyznaczenie kosztu aktualnej ścieżki
    actual_cost = self.travel_cost(matrix, actual_path)

    # Wyznaczenie najlepszej ścieżki
    best_path = actual_path

    # Wyznaczenie najlepszego kosztu
    best_path_cost = actual_cost

    # Lista zakazanych ruchów
    tabu_list = []

    # Aktualnie nie mamy najlepszej sąsiedniej ścieżki.
    best_neighbour_path = None

    # Koszt podróży dla najlepszej sąsiedniej ścieżki.
    best_neighbour_path_cost = float('inf')

    # Długość ścieżki
    path_length = len(actual_path)

    # Czas w [ms], po jakim znaleziono najlepsze rozwiązanie
    best_path_time = None

    # Wyznaczenie kryterium zakończenia algorytmu
    start_time = time.perf_counter_ns()
    end_time = start_time + int(maximal_time) * pow(10, 9)
```

```python
# Dopóki nie minie wyznaczony czas, wykonujemy następujące operacje.
while time.perf_counter_ns() < end_time:

    # Generujemy sąsiadów
    neighbour_list = self.generate_neighbours(path_length)

    # Zerujemy najlepszego sąsiada.
    best_neighbour = None

    # Dla każdego sąsiada w liście sąsiadów wykonujemy następujące operacje
    for neighbour in neighbour_list:

        # Wygenerowanie sąsiedniej ścieżki.
        if neighbour_type == "1":
            neighbour_path = self.swap_neighbours(neighbour, actual_path)
        elif neighbour_type == "2":
            neighbour_path = self.insert_neighbours(neighbour, actual_path)
        elif neighbour_type == "3":
            neighbour_path = self.invert_neighbours(neighbour, actual_path)
        else:
            neighbour_path = self.swap_neighbours(neighbour, actual_path)

        # Obliczamy koszt podróży sąsiednią ścieżką.
        neighbour_path_cost = self.travel_cost(matrix, neighbour_path)

        # Jeżeli sąsiednia ścieżka nie jest na tabu list lub jeżeli sąsiednia ścieżka jest bardzo
        # korzystna (kryterium aspiracji).
        if neighbour not in tabu_list or neighbour_path_cost < best_path_cost:

            # Usuwamy sąsiada z listy, jeżeli się na niej znajduje.
            if neighbour in tabu_list:
                tabu_list.remove(neighbour)

            # Zapisujemy ścieżkę oraz koszt, jeżeli są one najlepsze spośród reszty sąsiadów.
            if neighbour_path_cost < best_neighbour_path_cost:
                best_neighbour = neighbour
                best_neighbour_path = neighbour_path.copy()
                best_neighbour_path_cost = neighbour_path_cost
```

```
    # Jeżeli koszt najlepszego sąsiada jest mniejszy od najlepszego kosztu, to uznajemy tego sąsiada za najlepszego.
    if best_neighbour_path_cost < best_path_cost:
        best_path = best_neighbour_path.copy()
        best_path_cost = best_neighbour_path_cost
        best_path_time = (time.perf_counter_ns() - start_time) / pow(10, 6)
    else:
        iteration_without_improvement_counter = iteration_without_improvement_counter + 1

    # Jeżeli ilość iteracji bez polepszenia rozwiązania wynosi zadaną wartość, to generujemy nowe startowe rozwiązanie
    # (strategia dywersyfikacji).
    if iteration_without_improvement_counter == max_iteration_without_improvement:
        actual_path = self.start_path(matrix)
        iteration_without_improvement_counter = 0
    else:
        actual_path = best_neighbour_path.copy()

    # Dodanie ścieżki na zabronioną listę
    tabu_list.append(best_neighbour)

    # Jeżeli długość tabu listy jest większa od długości kadencji, to usuwamy z listy najstarszego sąsiada.
    if len(tabu_list) > cadence_length:
        tabu_list.pop(0)

# Dodajemy na koniec ścieżki miasto startowe.
best_path += (best_path[0],)

# Zwracamy najniższy koszt oraz najlepszą ścieżkę.
return best_path_cost, best_path, best_path_time
```

b) Opis najważniejszych klas w projekcie:

Klasa user_interface tworzy interfejs konsolowy oraz umozliwia wywoływanie poszczególnych funkcji programu takich jak wczytywanie grafu z pliku, uruchomienie algorytmu, ustawienie odpowiednich parametrów dla algorytmu czy też testy automatyczne.

Klasa graph_loader ma zaimplementowaną możliwość wczytywania plików tsp, ftv i rbg. Algorytm wczytujący graf automatycznie rozpoznaje, jaki typ pliku jest wczytywany i odpowiednie umieszcza wartości w macierzy.

Klasa automatic_test posiada funkcję uruchamiającą testy automatyczne, z których korzystałem robiąc sprawozdanie z projektu.

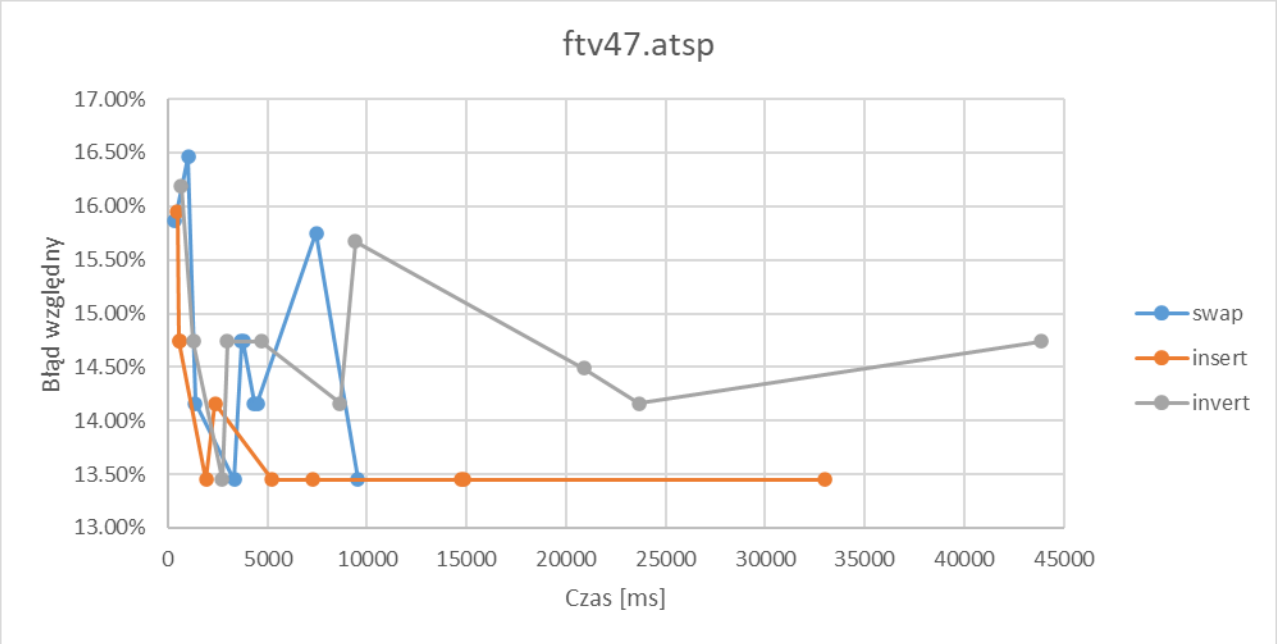
Klasa tabu_search_algorithm zawiera wszystkie funkcje bezpośrednio używane przez algorytm (m.in. algorytm tabu search, funkcję generującą rozwiązanie startowe metodą zachłanną oraz funkcje generujące sąsiadów).

c) Wyniki pomiarów:

typ sąsiedztwa	plik	optimum	czas podejmowania prób
swap	ftv47.atsp	1776	2 min
Powtórzenie	Wynik	Czas odnalezienia [ms]	Błąd względny
8	2111	329.4683	15.87%
2	2126	1027.6233	16.46%
6	2069	1362.1623	14.16%
7	2052	3319.3824	13.45%
10	2083	3678.9158	14.74%
5	2083	3767.6373	14.74%
4	2069	4324.8199	14.16%
3	2069	4482.905	14.16%
1	2108	7459.6235	15.75%
9	2052	9549.9532	13.45%
		średnia:	14.69%

typ sąsiedztwa	plik	optimum	czas podejmowania prób
insert	ftv47.atsp	1776	2 min
Powtórzenie	Wynik	Czas odnalezienia [ms]	Błąd względny
7	2113	473.2452	15.95%
8	2083	561.8914	14.74%
1	2083	574.2159	14.74%
9	2052	1904.2435	13.45%
2	2069	2379.1321	14.16%
6	2052	5224.4453	13.45%
5	2052	7288.7479	13.45%
3	2052	14722.0411	13.45%
10	2052	14873.9845	13.45%
4	2052	32983.2005	13.45%
		średnia:	14.03%

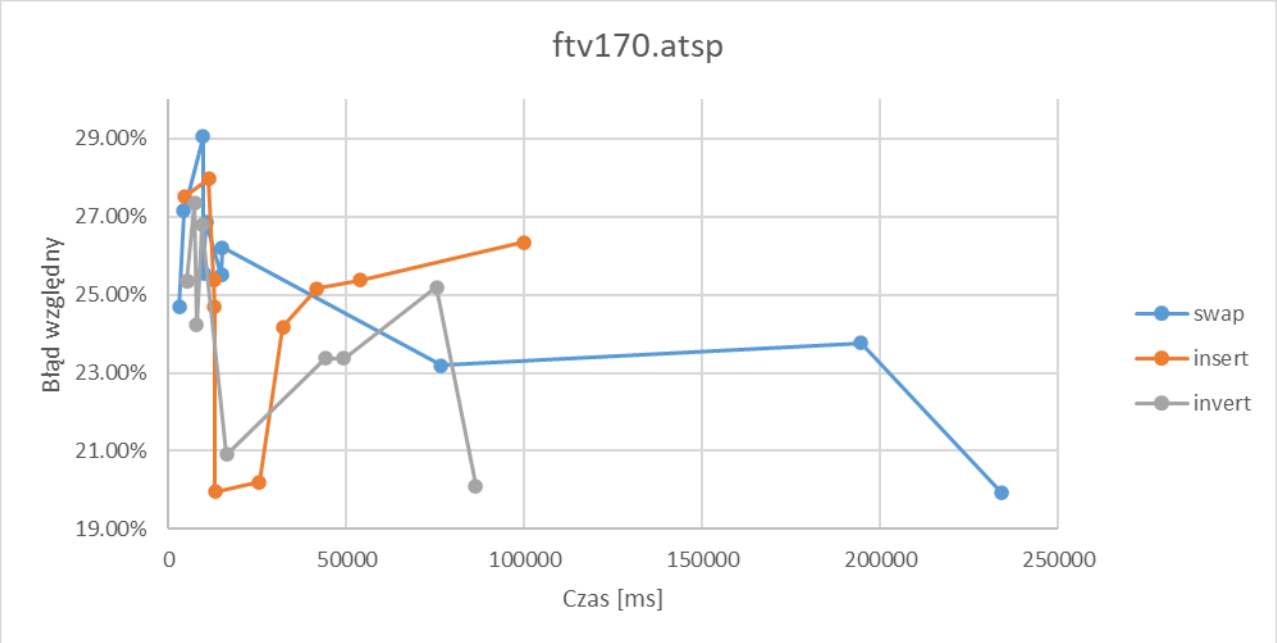
typ sąsiedztwa	plik	optimum	czas podejmowania prób
invert	ftv47.atsp	1776	2 min
Powtórzenie	Wynik	Czas odnalezienia [ms]	Błąd względny
10	2119	652.3957	16.19%
5	2083	1277.6152	14.74%
4	2052	2730.5941	13.45%
8	2083	2964.3225	14.74%
2	2083	4672.0441	14.74%
6	2069	8637.9948	14.16%
3	2106	9400.0142	15.67%
9	2077	20883.7663	14.49%
1	2069	23643.2311	14.16%
7	2083	43834.6837	14.74%
		średnia:	14.71%



typ sąsiedztwa	plik	optimum	czas podejmowania prób
swap	ftv170.atsp	2755	4 min
Powtórzenie	Wynik	Czas odnalezienia [ms]	Błąd względny
2	3658	3011.8886	24.69%
9	3782	4259.8022	27.15%
1	3883	9594.3091	29.05%
4	3701	9902.1981	25.56%
3	3767	10596.0443	26.86%
6	3698	14870.1771	25.50%
7	3734	15063.5191	26.22%
8	3587	76567.6416	23.19%
5	3614	194589.4734	23.77%
10	3440	234141.9306	19.91%
		średnia:	25.19%

typ sąsiedztwa	plik	optimum	czas podejmowania prób
insert	ftv170.atsp	2755	4 min
Powtórzenie	Wynik	Czas odnalezienia [ms]	Błąd względny
9	3801	4698.8817	27.52%
7	3825	11371.7339	27.97%
10	3692	12664.8352	25.38%
6	3658	12975.9488	24.69%
8	3442	13172.6031	19.96%
1	3452	25510.6009	20.19%
2	3633	32209.7314	24.17%
5	3681	41783.751	25.16%
3	3692	53716.1578	25.38%
4	3740	99835.633	26.34%
		średnia:	24.67%

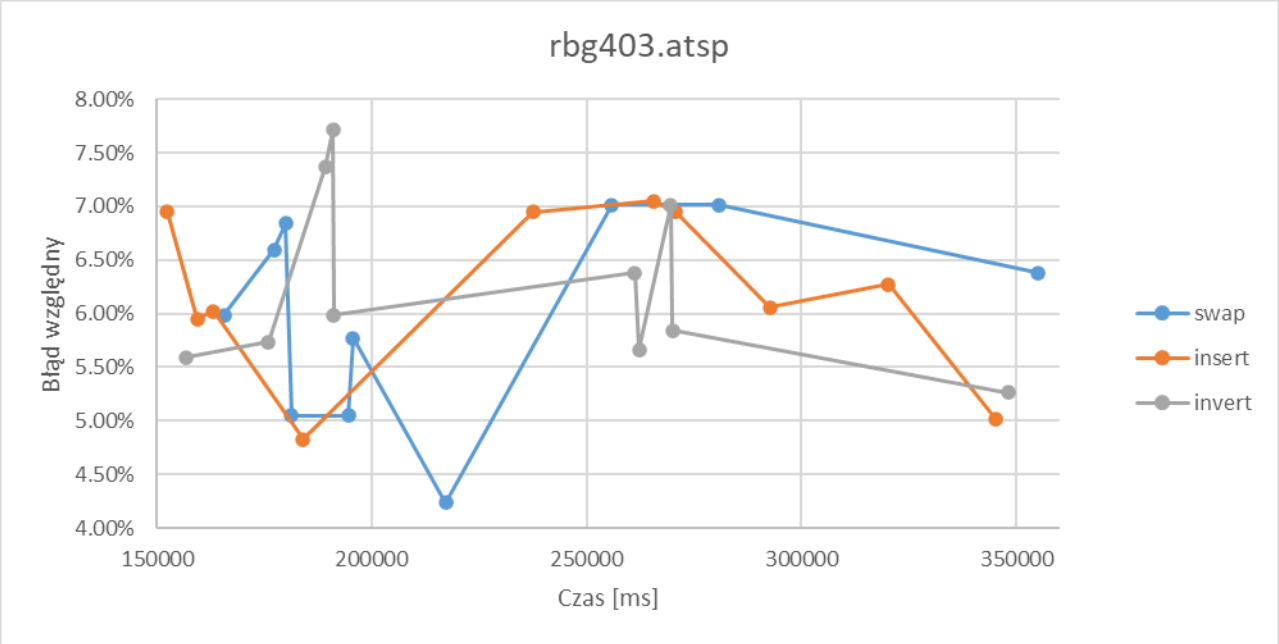
typ sąsiedztwa	plik	optimum	czas podejmowania prób
invert	ftv170.atsp	2755	4 min
Powtórzenie	Wynik	Czas odnalezienia [ms]	Błąd względny
6	3691	5124.5346	25.36%
9	3690	5260.4749	25.34%
3	3792	7357.7327	27.35%
2	3636	7903.164	24.23%
4	3763	9411.9624	26.79%
10	3483	16293.64	20.90%
7	3595	43949.1438	23.37%
1	3595	49264.1715	23.37%
8	3683	75407.1268	25.20%
5	3448	86240.8264	20.10%
		średnia:	24.20%



typ sąsiedztwa	plik	optimum	czas podejmowania prób
swap	rbg403	2465	6 min
Powtórzenie	Wynik	Czas odnalezienia [ms]	Błąd względny
1	2622	165678.4786	5.99%
8	2639	177256.3732	6.59%
9	2646	179970.8472	6.84%
2	2596	181408.1239	5.05%
5	2596	194589.4734	5.05%
7	2616	195608.5567	5.77%
4	2574	217240.6371	4.23%
3	2651	255887.908	7.02%
10	2651	280755.5121	7.02%
6	2633	355035.2593	6.38%
		średnia:	5.99%

typ sąsiedztwa	plik	optimum	czas podejmowania prób
insert	rbg403	2465	6 min
Powtórzenie	Wynik	Czas odnalezienia [ms]	Błąd względny
1	2649	152421.2628	6.95%
8	2621	159382.3386	5.95%
3	2623	163116.3562	6.02%
4	2590	184078.2608	4.83%
5	2649	237510.0988	6.95%
9	2652	265551.4707	7.05%
10	2649	270741.3722	6.95%
2	2624	292678.5378	6.06%
6	2630	320267.9241	6.27%
7	2595	345255.87	5.01%
		średnia:	6.20%

typ sąsiedztwa	plik	optimum	czas podejmowania prób
invert	rbg403	2465	6 min
Powtórzenie	Wynik	Czas odnalezienia [ms]	Błąd względny
2	2611	156945.2744	5.59%
5	2615	176045.1399	5.74%
4	2661	189205.9104	7.37%
10	2671	191074.7874	7.71%
6	2622	191168.3994	5.99%
3	2633	261172.2878	6.38%
1	2613	262340.5859	5.66%
7	2651	269629.0967	7.02%
8	2618	270009.1415	5.84%
9	2602	348067.1375	5.27%
		średnia:	6.26%



e) Najlepsze ścieżki dla każdego z plików:

1) ftv47.atsp (optymalny koszt: 1776):

swap: (koszt: 2052, ścieżka: [34, 13, 46, 36, 14, 16, 45, 20, 38, 18, 17, 12, 32, 6, 24, 4, 3, 33, 27, 28, 2, 41, 43, 22, 40, 47, 26, 42, 1, 0, 25, 10, 11, 8, 9, 29, 30, 5, 31, 7, 23, 35, 15, 19, 44, 39, 21, 37, 34])

insert: (koszt: 2052, ścieżka: [34, 13, 46, 36, 14, 16, 45, 20, 38, 18, 17, 12, 32, 6, 24, 4, 3, 33, 27, 28, 2, 41, 43, 22, 40, 47, 26, 42, 0, 25, 1, 10, 11, 8, 9, 29, 30, 5, 31, 7, 23, 35, 15, 19, 44, 39, 21, 37, 34])

invert: (koszt: 2052, ścieżka: [13, 34, 46, 36, 14, 16, 45, 20, 38, 18, 17, 12, 32, 6, 24, 4, 3, 33, 27, 28, 2, 41, 43, 22, 40, 47, 26, 42, 0, 25, 1, 10, 11, 8, 9, 29, 30, 5, 31, 7, 23, 35, 15, 19, 44, 39, 21, 37, 13])

2) ftv170.atsp (optymalny koszt: 2755)

swap: (koszt: 3440 , ścieżka: [144, 143, 141, 134, 131, 113, 164, 127, 126, 125, 124, 121, 120, 122, 123, 162, 102, 103, 117, 118, 119, 129, 128, 130, 135, 138, 139, 140, 6, 7, 8, 9, 10, 76, 74, 75, 11, 12, 18, 19, 20, 21, 22, 23, 27, 28, 29, 30, 31, 33, 35, 34, 156, 40, 39, 38, 37, 49, 170, 73, 77, 1, 2, 0, 81, 80, 79, 82, 78, 72, 71, 60, 50, 51, 52, 53, 43, 55, 54, 58, 59, 61, 68, 67, 167, 70, 87, 85, 86, 83, 84, 69, 66, 63, 64, 56, 57, 62, 65, 88, 153, 154, 89, 90, 91, 94, 96, 97, 99, 98, 95, 92, 93, 166, 108, 107, 106, 105, 165, 163, 100, 101, 104, 110, 109, 114, 115, 116, 136, 137, 147, 148, 149, 161, 152, 151, 14, 13, 17, 32, 158, 36, 157, 41, 155, 42, 45, 44, 46, 47, 48, 168, 3, 4, 5, 133, 169, 111, 112, 132, 142, 160, 150, 26, 24, 15, 159, 16, 25, 146, 145, 144])

insert: (koszt: 3442, ścieżka: [144, 143, 142, 141, 131, 113, 164, 127, 126, 125, 124, 121, 120, 122, 123, 162, 102, 103, 117, 118, 119, 129, 128, 130, 135, 138, 139, 140, 6, 7, 8, 9, 10, 76, 74, 75, 11, 12, 18, 19, 20, 21, 22, 23, 26, 27, 28, 30, 31, 33, 35, 34, 156, 40, 39, 38, 37, 49, 170, 73, 77, 1, 2, 0, 81, 80, 79, 82, 78, 72, 71, 60, 50, 51, 52, 53, 43, 55, 54, 58, 59, 61, 68, 67, 167, 70, 87, 85, 86, 83, 84, 69, 66, 63, 64, 56, 57, 62, 65, 88, 153, 154, 89, 90, 91, 94, 96, 97, 99, 98, 95, 92, 93, 166, 108, 107, 106, 105, 165, 163, 100, 101, 104, 114, 110, 109, 115, 116, 136, 137, 147, 148, 149, 150, 161, 160, 14, 13, 17, 32, 158, 36, 157, 41, 155, 42, 45, 44, 46, 47, 48, 168, 3, 4, 5, 133, 169, 111, 112, 132, 134, 152, 151, 29, 24, 15, 159, 16, 25, 146, 145, 144])

invert: (koszt: 3448, ścieżka: [150, 160, 151, 152, 142, 141, 134, 131, 113, 164, 127, 126, 125, 124, 121, 120, 122, 123, 162, 102, 103, 117, 118, 119, 129, 128, 130, 135, 138, 139, 140, 6, 7, 8, 9, 10, 76, 74, 75, 11, 12, 18, 19, 20, 32, 22, 23, 26, 27, 28, 29, 30, 31, 35, 34, 156, 40, 39, 38, 37, 49, 170, 73, 77, 1, 2, 0, 81, 80, 79, 82, 78, 72, 71, 60, 50, 51, 52, 53, 43, 55, 54, 58, 59, 61, 68, 67, 167, 70, 87, 85, 86, 83, 84, 69, 66, 63, 64, 56, 57, 62, 65, 88, 153, 154, 89, 90, 91, 94, 96, 97, 165, 163, 99, 98, 95, 92, 93, 166, 108, 107, 106, 105, 104, 110, 109, 114, 115, 116, 100, 101, 136, 137, 147, 148, 149, 161, 14, 13, 17, 158, 36, 157, 33, 41, 155, 42, 45, 44, 46, 47, 48, 168, 3, 4, 5, 133, 169, 111, 112, 132, 143, 144, 146, 145, 15, 159, 16, 21, 24, 25, 150])

3) rbg403.atsp (optymalny koszt: 2465):

swap: (koszt: 2574 , ścieżka: [23, 14, 62, 13, 205, 204, 24, 36, 32, 274, 46, 33, 376, 68, 38, 270, 19, 18, 402, 287, 83, 43, 34, 295, 50, 304, 394, 225, 58, 8, 6, 163, 3, 2, 61, 107, 386, 47, 112, 322, 257, 308, 5, 312, 35, 217, 84, 272, 28, 129, 360, 110, 314, 59, 153, 310, 29, 77, 31, 52, 40, 39, 78, 226, 147, 79, 69, 245, 81, 22, 21, 82, 247, 249, 172, 26, 246, 367, 75, 160, 15, 333, 267, 281, 221, 67, 66, 60, 44, 88, 96, 94, 90, 72, 57, 164, 25, 91, 92, 51, 49, 37, 251, 301, 120, 392, 351, 293, 93, 145, 101, 180, 165, 316, 364, 303, 71, 97, 387, 349, 73, 99, 389, 111, 369, 237, 124, 269, 355, 115, 114, 353, 263, 244, 80, 209, 341, 313, 20, 340, 169, 27, 239, 117, 278, 122, 119, 168, 142, 150, 384, 383, 361, 146, 144, 105, 391, 154, 155, 7, 393, 372, 356, 328, 317, 108, 256, 182, 109, 275, 210, 265, 326, 258, 140, 374, 363, 289, 123, 192, 189, 170, 284, 290, 125, 74, 214, 401, 216, 200, 198, 159, 323, 305, 215, 309, 162, 48, 358, 327, 234, 228, 224, 151, 271, 350, 307, 143, 285, 282, 357, 241, 113, 395, 132, 243, 175, 291, 315, 161, 148, 325, 134, 306, 319, 135, 280, 279, 264, 268, 16, 388, 254, 223, 344, 345, 173, 354, 342, 266, 141, 116, 385, 300, 302, 118, 149, 347, 352, 104, 9, 236, 227, 381, 188, 299, 298, 63, 297, 130, 178, 100, 138, 337, 181, 346, 338, 331, 378, 398, 396, 156, 390, 157, 1, 176, 318, 195, 179, 368, 366, 292, 329, 294, 330, 343, 136, 370, 380, 379, 230, 382, 212, 103, 336, 321, 362, 208, 85, 252, 371, 177, 283, 277, 375, 191, 187, 186, 53, 400, 262, 261, 193, 190, 86, 11, 106, 359, 133, 201, 89, 158, 339, 126, 55, 399, 377, 286, 273, 335, 207, 45, 184, 4, 131, 242, 70, 397, 260, 56, 232, 127, 213, 41, 64, 365, 166, 199, 373, 311, 334, 102, 276, 259, 128, 255, 194, 324, 320, 95, 10, 183, 17, 12, 332, 296, 174, 185, 167, 196, 0, 202, 206, 30, 222, 250, 42, 152, 76, 220, 197, 87, 137, 54, 231, 288, 233, 235, 121, 211, 98, 238, 229, 240, 219, 65, 218, 203, 248, 348, 171, 253, 139, 23])

insert: (koszt: 2590, ścieżka: [25, 367, 75, 10, 27, 11, 59, 310, 6, 23, 14, 62, 13, 205, 204, 24, 36, 32, 274, 46, 33, 376, 68, 38, 270, 19, 18, 402, 287, 83, 43, 34, 295, 65, 304, 394, 225, 58, 8, 29, 163, 3, 2, 61, 107, 386, 47, 112, 264, 330, 308, 5, 126, 150, 9, 84, 272, 28, 232, 129, 221, 300, 77, 31, 52, 40, 39, 78, 226, 147, 79, 69, 333, 267, 281, 81, 22, 21, 82, 247, 103, 172, 26, 314, 159, 7, 238, 166, 67, 66, 60, 44, 88, 96, 94, 90, 72, 57, 91, 180, 165, 387, 92, 51, 49, 37, 42, 301, 120, 392, 351, 293, 93, 145, 218, 302, 118, 340, 167, 158, 269, 360, 353, 263, 364, 303, 71, 131, 217, 349, 73, 246, 262, 261, 196, 0, 355, 115, 114, 396, 177, 209, 278, 122, 119, 168, 239, 117, 251, 328, 317, 104, 203, 384, 383, 361, 146, 144, 105, 391, 154, 111, 369, 182, 374, 363, 289, 128, 255, 362, 265, 275, 210, 110, 326, 258, 318, 286, 74, 214, 192, 189, 162, 48, 358, 327, 170, 284, 290, 125, 273, 350, 313, 323, 216, 200, 198, 375, 191, 187, 234, 228, 224, 240, 219, 357, 322, 370, 237, 151, 101, 285, 282, 257, 132, 243, 288, 291, 315, 161, 148, 325, 134, 306, 305, 215, 309, 135, 280, 279, 236, 227, 320, 137, 54, 254, 223, 319, 345, 173, 344, 208, 311, 385, 199, 229, 393, 149, 347, 401, 354, 342, 266, 124, 339, 85, 171, 63, 297, 153, 248, 329, 294, 178, 138, 337, 142, 176, 156, 390, 181, 346, 338, 331, 378, 398, 194, 324, 381, 188, 299, 298, 190, 100, 98, 127, 399, 377, 195, 108, 256, 222, 185, 64, 365, 242, 70, 244, 80, 184, 4, 186, 53, 207, 45, 372, 356, 102, 95, 55, 253, 276, 259, 316, 249, 152, 76, 352, 130, 335, 206, 30, 336, 321, 379, 201, 89, 366, 292, 343, 373, 193, 283, 277, 341, 388, 348, 133, 157, 1, 211, 160, 15, 123, 106, 359, 86, 220, 197, 140, 334, 174, 250, 175, 109, 312, 35, 169, 271, 155, 143, 183, 17, 12, 268, 16, 97, 230, 382, 212, 87, 202, 400, 139, 179, 368, 245, 213, 41, 136, 99, 389, 332, 296, 307, 20, 233, 380, 241, 113, 395, 50, 56, 231, 235, 121, 397, 260, 141, 116, 252, 371, 164, 25])

invert: (koszt: 2602, ścieżka: [23, 14, 62, 13, 205, 204, 24, 36, 32, 274, 46, 33, 376, 68, 38, 270, 19, 18, 402, 287, 83, 43, 34, 295, 50, 56, 367, 75, 58, 8, 6, 163, 3, 2, 61, 107, 386, 47, 112, 322, 257, 65, 312, 35, 176, 93, 84, 272, 28, 360, 150, 111, 369, 155, 143, 310, 29, 77, 31, 52, 40, 39, 78, 226, 147, 79, 69, 245, 81, 22, 21, 82, 247, 249, 172, 26, 314, 59, 190, 233, 133, 1, 267, 281, 221, 67, 66, 60, 44, 88, 96, 94, 90, 72, 57, 64, 15, 153, 92, 51, 49, 37, 151, 301, 120, 392, 351, 293, 220, 197, 159, 7, 183, 17, 364, 303, 71, 242, 70, 349, 73, 99, 389, 106, 165, 316, 256, 131, 355, 115, 114, 353, 263, 102, 91, 372, 356, 400, 302, 118, 11, 231, 288, 209, 278, 122, 119, 168, 104, 9, 384, 383, 361, 146, 144, 105, 391, 154, 169, 27, 184, 4, 251, 113, 317, 108, 255, 237, 109, 275, 210, 265, 326, 258, 207, 374, 363, 289, 123, 192, 189, 124, 284, 290, 125, 74, 214, 401, 216, 200, 198, 307, 323, 305, 215, 309, 162, 48, 358, 327, 234, 228, 224, 240, 219, 195, 86, 101, 285, 282, 357, 328, 370, 98, 132, 243, 175, 291, 315, 161, 148, 325, 134, 306, 319, 135, 280, 279, 136, 395, 211, 320, 254, 223, 344, 345, 173, 354, 342, 266, 139, 87, 385, 199, 152, 76, 149, 347, 352, 166, 300, 236, 227, 381, 103, 177, 171, 63, 297, 232, 178, 100, 138, 337, 181, 346, 338, 331, 378, 398, 396, 156, 390, 157, 339, 164, 25, 271, 350, 182, 366, 292, 329, 294, 330, 343, 235, 121, 380, 379, 110, 42, 252, 371, 336, 321, 362, 142, 174, 250, 97, 95, 283, 277, 375, 191, 187, 318, 299, 298, 12, 304, 260, 308, 394, 225, 313, 20, 373, 201, 89, 340, 196, 0, 335, 332, 296, 217, 239, 117, 246, 126, 55, 170, 334, 130, 387, 145, 388, 206, 30, 141, 116, 140, 160, 365, 333, 167, 180, 359, 129, 85, 193, 268, 16, 222, 185, 179, 368, 213, 41, 45, 248, 348, 188, 276, 259, 194, 324, 212, 311, 10, 158, 269, 244, 80, 341, 186, 53, 218, 203, 286, 273, 230, 382, 262, 261, 128, 264, 137, 54, 238, 229, 393, 241, 253, 208, 397, 5, 202, 127, 399, 377, 23])

f) Wnioski:

Algorytm tabu search dla asymetrycznego problemu komiwojażera wydaje się być dobrym rozwiązaniem, nie gwarantuje on optymalnych wyników, jednak zazwyczaj wyniki są do optimum zbliżone.

Wszystkie 3 definicje sąsiedztwa dają podobny błąd dla algorytmu tabu search, ciężko więc wyłonić z ich trójki najlepszy. Wykresy zależności błędu od czasu nie mają regularnego kształtu co może być związane z tym, że sąsiedzi byli dobierani w sposób losowy.