Sprawozdanie z implementacji heurystyk przeszukiwania lokalnego dla zmodyfikowanego problemu komiwojażera

Filip Rosiak 151799 Eryk Stec 152948

30 marca 2025

Streszczenie

W niniejszym sprawozdaniu przedstawiono implementację i analizę heurystyk przeszukiwania lokalnego dla zmodyfikowanego problemu komiwojażera. Problem polega na ułożeniu dwóch rozłącznych zamkniętych ścieżek, każda zawierająca około 50% wierzchołków, minimalizując łączną długość obu ścieżek. Zaimplementowano i porównano metodę przeszukiwania w wersji stromej i zachłannej, startując z rozwiązań losowych i zachłannych, z dwoma rodzajami sąsiedztwa (międzytrasowa wymiana wierzchołków i wewnątrztrasowa wymiana krawędzi albo wierzchołków). Eksperymenty przeprowadzono na instancjach kroa200 i krob200 z biblioteki TSPLib.

1 Opis problemu

Rozważany problem jest modyfikacją klasycznego problemu komiwojażera. Dany jest zbiór wierzchołków i symetryczna macierz odległości pomiędzy dowolną parą wierzchołków. Zadanie polega na ułożeniu dwóch rozłącznych zamkniętych ścieżek (cykli), każda zawierająca 50% wierzchołków (jeżeli liczba wierzchołków w instancji nie jest parzysta, to pierwsza ścieżka zawiera jeden wierzchołek więcej), minimalizując łączną długość obu ścieżek.

Do testów wykorzystano instancje `kroa200` i `krob200` z biblioteki TSPLib. Są to dwuwymiarowe instancje euklidesowe, gdzie dla każdego wierzchołka podane są dwie współrzędne, a odległość pomiędzy wierzchołkami jest odległością euklidesową zaokrąglaną do liczby całkowitej.

2 Zaimplementowane algorytmy rozwiązań startowych

W ramach zadania zaimplementowano następujące heurystyki konstrukcyjne:

2.1 Algorytm najbliższego sąsiada

Algorytm jest inspirowany metodą najbliższego sąsiada dla klasycznego problemu komiwojażera, dostosowany do rozważanego problemu z dwoma cyklami. Pseudokod algorytmu przedstawiono w Algorytmie 1.

Algorytm 1 Algorytm najbliższego sąsiada dla zmodyfikowanego problemu komiwojażera

Znajdź punkty startowe:
Losowo wybierz wierzchołki (s_1, s_2)
Te wierzchołki będą punktami startowymi dla cykli C_1 i C_2
Inicjalizuj cykle:
$C_1 = [s_1]$
$C_2 = [s_2]$
Dostępne wierzchołki $A = \{0, 1, ..., n-1\} \setminus \{s_1, s_2\}$
Naprzemiennie rozbudowuj cykle:
while A nie jest puste **do**
    Znajdź wierzchołek $v \in A$ najbliższy ostatniemu wierzchołkowi w C_1
    Dodaj v do C_1
    Usuń v z A
    if A nie jest puste **then**
        Znajdź wierzchołek $v \in A$ najbliższy ostatniemu wierzchołkowi w C_2
        Dodaj v do C_2
        Usuń v z A
    end if
end while
return (C_1, C_2)

2.2 Rozwiązanie losowe

Algorytm generuje cykle z losowo wybranymi wierzchołkami 2.

Algorytm 2 Algorytm inicjalizacji losowej dla zmodyfikowanego problemu komiwojażera

1: **Inicjalizuj wierzchołki:**
2: $A = \{0, 1, ..., n-1\}$
3: **Potasuj wierzchołki:**
4: Potasuj elementy zbioru A
5: **Podziel wierzchołki na dwa cykle:**
6: $C_1 = A[0 : \lfloor n/2 \rfloor]$
7: $C_2 = A[\lfloor n/2 \rfloor : n]$
8: **return** (C_1, C_2)

3 Zaimplementowane algorytmy generowania sąsiedztwa

W ramach zadania zaimplementowano następujące algorytmy:

3.1 Wymiana wierzchołków

Algorytm generuje sąsiedztwo zawierające wszystkie możliwe wymiany międzytrasowe wierzchołków oraz wymiany wewnątrztrasowe wierzchołków 3.

Algorytm 3 Generowanie ruchów - Wymiana wierzchołków

Inicjalizuj sąsiedztwo
$S = []$
Generowanie ruchów między cyklami:
for $i = 0$ to $|C_1| - 1$ **do**
    for $j = 0$ to $|C_2| - 1$ **do**
        Oblicz koszt wymiany wierzchołków i i j pomiędzy cyklami C_1 i C_2:
        Oblicz nowe koszty cykli
        Dodaj ruch do sąsiedztwa S
    end for
end for
Generowanie ruchów w obrębie cykli:
for $cycle\_num = 0$ to 1 **do**
    for $i = 0$ to $|C_{cycle\_num}| - 1$ **do**
        for $j = i$ to $|C_{cycle\_num}| - 1$ **do**
            Oblicz koszt wymiany wierzchołków w cyklu C_{cycle_num}:
            Oblicz nowy koszt cyklu
            Dodaj ruch do sąsiedztwa S
        end for
    end for
end for
return S

3.2 Wymiana krawędzi

Algorytm generuje sąsiedztwo zawierające wszystkie możliwe wymiany międzytrasowe wierzchołków oraz wymiany wewnątrztrasowe krawędzi 4.

Algorytm 4 Generowanie ruchów - Wymiana krawędzi

Inicjalizuj sąsiedztwo
$S = []$
Generowanie ruchów między cyklami:
for $i = 0$ to $|C_1| - 1$ **do**
    for $j = 0$ to $|C_2| - 1$ **do**
        Oblicz koszt wymiany wierzchołków i i j pomiędzy cyklami C_1 i C_2:
        Oblicz nowe koszty cykli
        Dodaj ruch do sąsiedztwa S
    end for
end for
Generowanie ruchów w obrębie cykli:
for $cycle\_num = 0$ to 1 **do**
    for $i = 0$ to $|C_{cycle\_num}| - 1$ **do**
        for $j = i$ to $|C_{cycle\_num}| - 1$ **do**
            Oblicz koszt wymiany krawędzi w cyklu C_{cycle_num}:
            Oblicz nowy koszt cyklu
            Dodaj ruch do sąsiedztwa S
        end for
    end for
end for
return S

4 Zaimplementowane algorytmy przeszukiwania lokalnego

W ramach zadania zaimplementowano następujące przeszukiwania lokalnego:

4.1 Algorytm Steepest Local Search

Algorytm Steepest Local Search jest metodą przeszukiwania lokalnego, w której iteracyjnie generowane są możliwe ruchy w przestrzeni rozwiązań, a następnie wybierany jest najlepszy dostępny ruch prowadzący do najmniejszego kosztu. Algorytm kontynuuje proces, dopóki nie będzie już dostępny ruch prowadzący do lepszego rozwiązania. Pseudokod algorytmu przedstawiono w Algorytmie 5.

Algorytm 5 Algorytm Steepest Local Search

```
1: best_cycles = [C_1.copy(), C_2.copy()]
2: best_costs = calculate_total_cost(C_1, C_2, distance_matrix)
3: while True do
4:     Generuj sąsiedztwo S
5:     Wybierz ruch o najmniejszym koszcie
6:     if best_move_costs ≥ best_costs then
7:         Zakończ: Przerwij algorytm
8:     end if
9:     Aktualizacja kosztów i cykli:
10:    what_swap, cycle_num, i, j, new_costs = best_move
11:    best_costs = best_move_costs
12:    if what_swap = "vertices" then
13:        Wykonaj wymianę wierzchołków
14:    else if what_swap = "edges" then
15:        Wykonaj wymianę krawędzi
16:    else
17:        Wykonaj wymianę między cyklami
18:    end if
19: end while
20: return best_cycles, best_costs
```

4.2 Algorytm Greedy Local Search

Algorytm Greedy Local Search to metoda przeszukiwania lokalnego, w której generowane są ruchy w przestrzeni rozwiązań, a następnie wybierany jest pierwszy ruch, który prowadzi do poprawy rozwiązania. Proces powtarza się, aż nie będzie już możliwe znalezienie lepszego rozwiązania w sąsiedztwie 6.

Algorytm 6 Algorytm Greedy Local Search

```
1: best_cycles = [C_1.copy(), C_2.copy()]
2: best_costs = calculate_total_cost(C_1, C_2, distance_matrix)
3: while True do
4:     Generuj sąsiedztwo S
5:     Losowo przetasuj ruchy w sąsiedztwie          ▷ random.shuffle()
6:     Próba poprawy rozwiązania:
7:     improved = False
8:     for each move in S do
9:         what_swap, cycle_num, i, j, new_costs = move
10:        if new_costs < best_costs then
11:            best_costs = new_costs
12:            if what_swap = "vertices" then
13:                Wykonaj wymianę wierzchołków
14:            else if what_swap = "edges" then
15:                Wykonaj wymianę krawędzi
16:            else
17:                Wykonaj wymianę między cyklami
18:            end if
19:            improved = True
20:            Przerwij iterację w celu ponownego wygenerowania sąsiedztwa
21:        end if
22:    end for
23:    if not improved then
24:        Zakończ: Przerwij algorytm
25:    end if
26: end while
27: return best_cycles, best_costs
```

4.3 Random Walk

Algorytm Random Walk to metoda przeszukiwania losowego, w której wykonywane są losowe ruchy w przestrzeni rozwiązań, a algorytm stara się znaleźć

rozwiązanie o najniższym koszcie. W każdej iteracji wykonywany jest losowy ruch będący wymianą między cyklami lub wewnątrz cykli 7.

Algorytm 7 Algorytm Random Walk

```
start_time = time()
best_cycles = [C_1, C_2]
best_costs = calculate_total_cost(C_1, C_2, distance_matrix)
cycles = [C_1, C_2]
costs = best_costs.copy()
while time() − start_time < time_limit do
    if random() < 0.5 then
        Wykonaj wymianę między cyklami
        if costs < best_costs then
            best_cycles = cycles
            best_costs = costs
        end if
    else
        Wykonaj wymianę wewnątrz cykli
        cycle_num = randint(0, 1)
        i, j = random.sample(range(|C_cycle_num|), 2)
        if random() < 0.5 then
            Wykonaj wymianę wierzchołków
        else
            Wykonaj wymianę krawędzi
        end if
        Oblicz nowy koszt
        if costs < best_costs then
            best_cycles = cycles
            best_costs = costs
        end if
    end if
end while
return best_cycles, best_costs
```

5 Wyniki eksperymentów

Każdy algorytm został uruchomiony 100 razy na instancjach kroa200 i krob200 dla każdej kombinacji. Poniżej przedstawiono wyniki eksperymentów.

Tabela 1: Wyniki eksperymentów dla instancji kroa200 i krob200

Kombinacja	**kroa200**	**krob200**
Steepest_vertices_random	73584 (63783 - 85227)	71369 (57133 - 87696)
Steepest_vertices_greedy	39135 (34563 - 45919)	39302 (35937 - 42724)
Steepest_edges_random	38619 (35788 - 41799)	38784 (37002 - 40703)
Steepest_edges_greedy	34214 (31474 - 37108)	34752 (31941 - 37619)
Greedy_vertices_random	66869 (56854 - 82352)	65570 (55638 - 74526)
Greedy_vertices_greedy	39280 (35094 - 45765)	39370 (35862 - 42657)
Greedy_edges_random	38778 (36125 - 41088)	38747 (36277 - 40721)
Greedy_edges_greedy	34979 (31636 - 37717)	35432 (32078 - 38777)
Random_walk	281003 (271676 - 285587)	276016 (270202 - 281053)

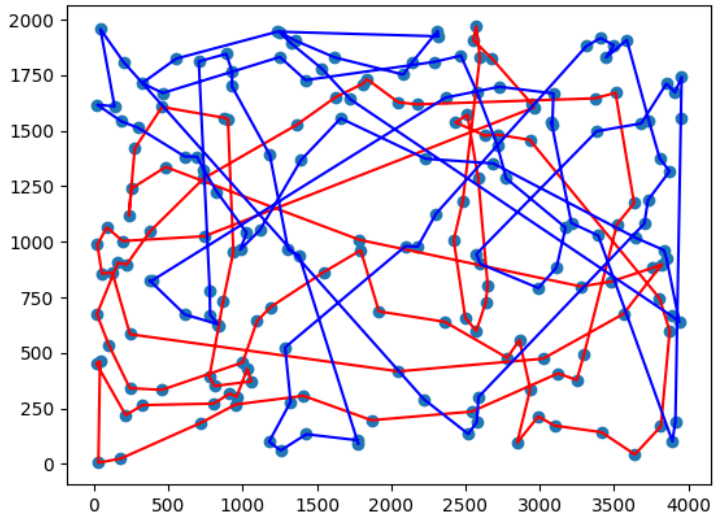
Tabela 2: Czasy eksperymentów dla instancji kroa200 i krob200

Kombinacja	**kroa200**	**krob200**
Steepest_vertices_random	4.90 (4.08 - 5.91)	5.16 (4.08 - 6.35)
Steepest_vertices_greedy	0.42 (0.20 - 0.63)	0.36 (0.09 - 0.63)
Steepest_edges_random	3.74 (3.32 - 4.16)	3.89 (3.36 - 4.68)
Steepest_edges_greedy	0.81 (0.58 - 1.21)	0.75 (0.53 - 1.13)
Greedy_vertices_random	17.62 (15.33 - 20.62)	18.47 (15.47 - 23.48)
Greedy_vertices_greedy	0.77 (0.25 - 1.59)	0.59 (0.11 - 1.07)
Greedy_edges_random	15.79 (14.07 - 18.07)	16.56 (14.42 - 18.82)
Greedy_edges_greedy	1.78 (1.16 - 2.79)	1.57 (0.86 - 2.62)
Random_walk	17.62 (17.62 - 17.62)	18.47 (18.47 - 18.47)

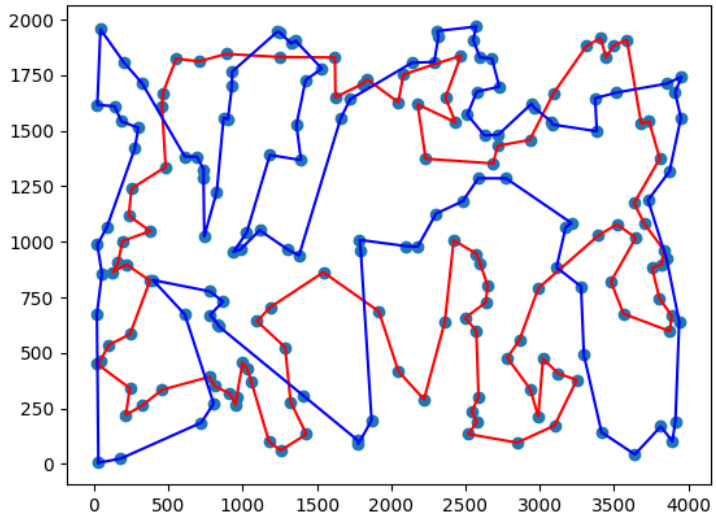
6 Wizualizacje

Poniżej przedstawiono wizualizacje najlepszych rozwiązań dla każdej kombinacji i instancji.

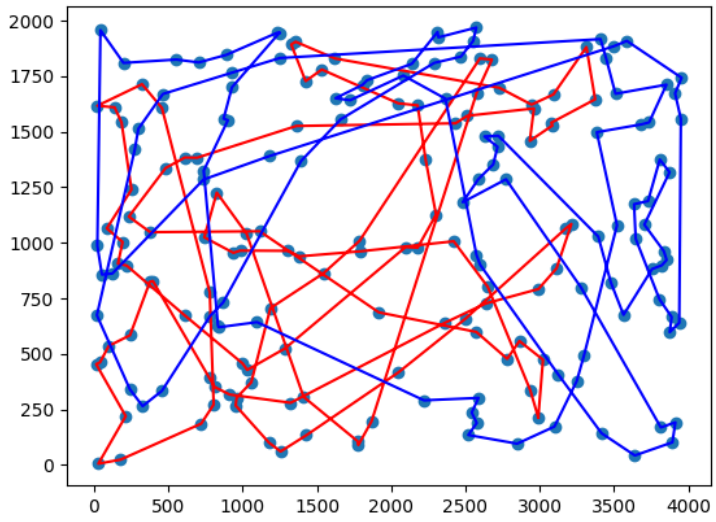
6.1 Instancja kroA200



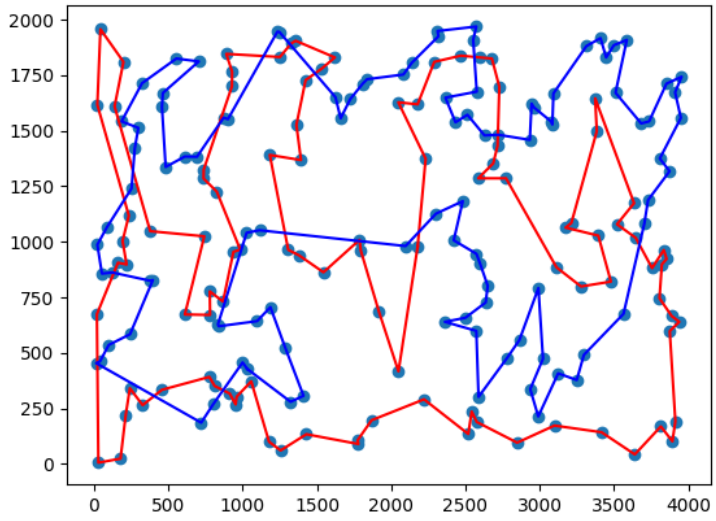
Rysunek 1: Rozwiązanie dla Steepest_vertices_random na instancji kroA200



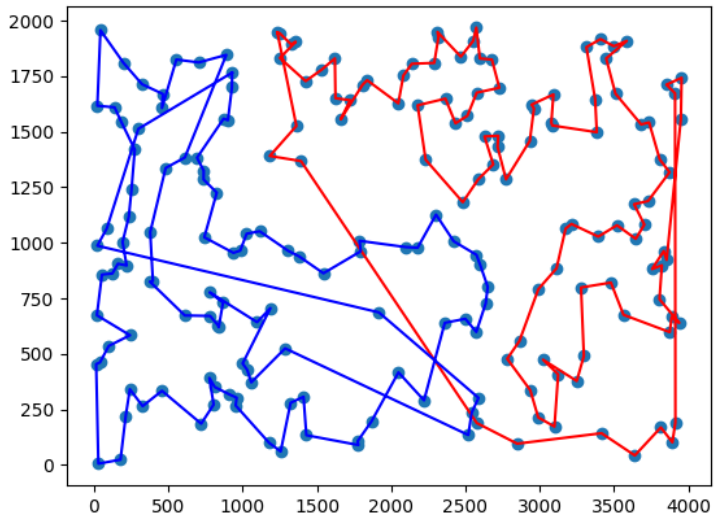
Rysunek 2: Rozwiązanie dla Steepest_edge_random na instancji kroA200



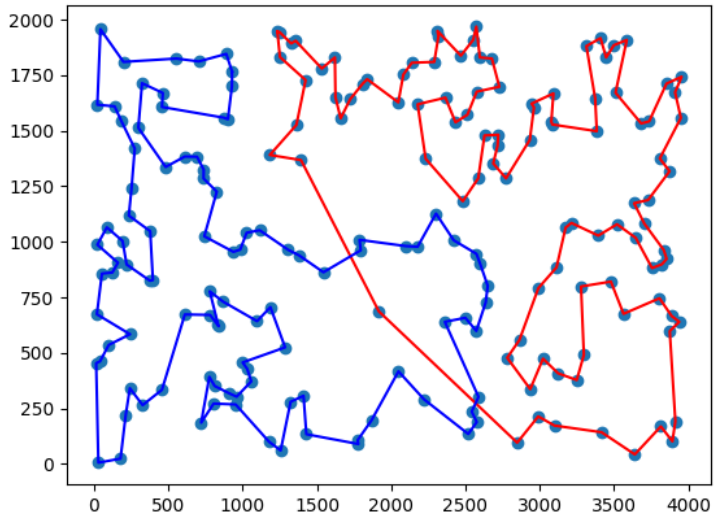
Rysunek 3: Rozwiązanie dla Greedy_vertices_random na instancji kroA200



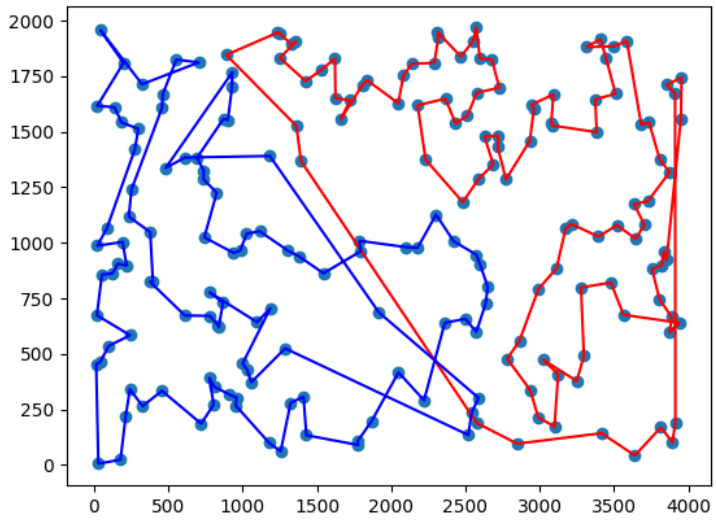
Rysunek 4: Rozwiązanie dla Greedy_edge_random na instancji kroA200



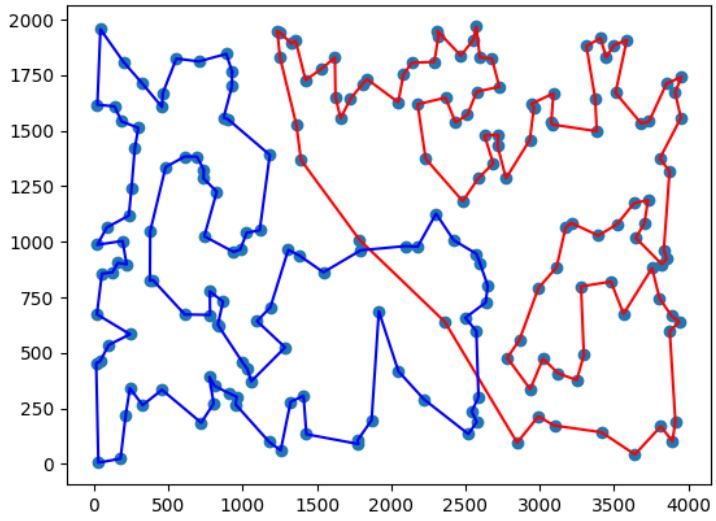
Rysunek 5: Rozwiązanie dla Steepest_vertices_greedy na instancji kroA200



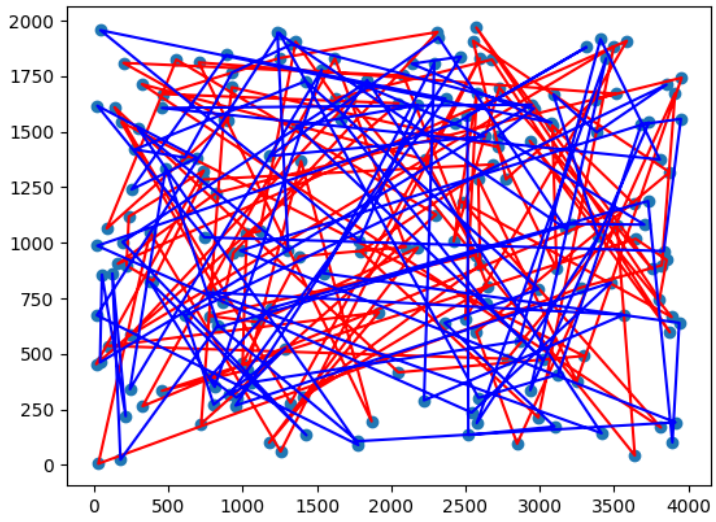
Rysunek 6: Rozwiązanie dla Steepest_edge_greedy na instancji kroA200



Rysunek 7: Rozwiązanie dla Greedy_vertices_greedy na instancji kroA200

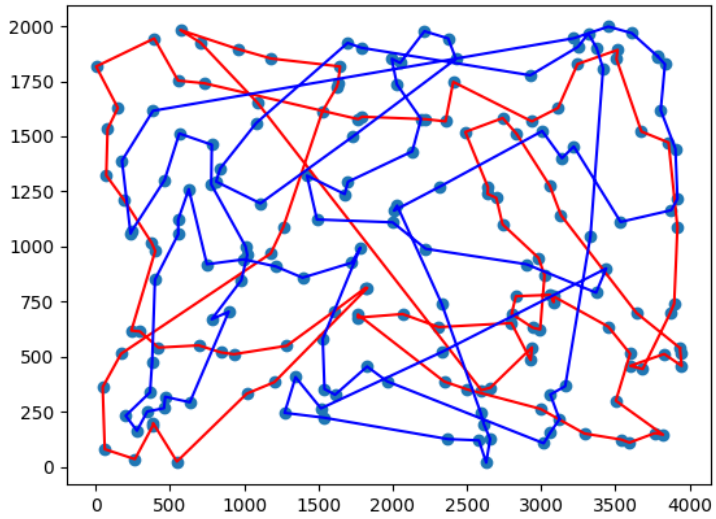


Rysunek 8: Rozwiązanie dla Greedy_edge_greedy na instancji kroA200

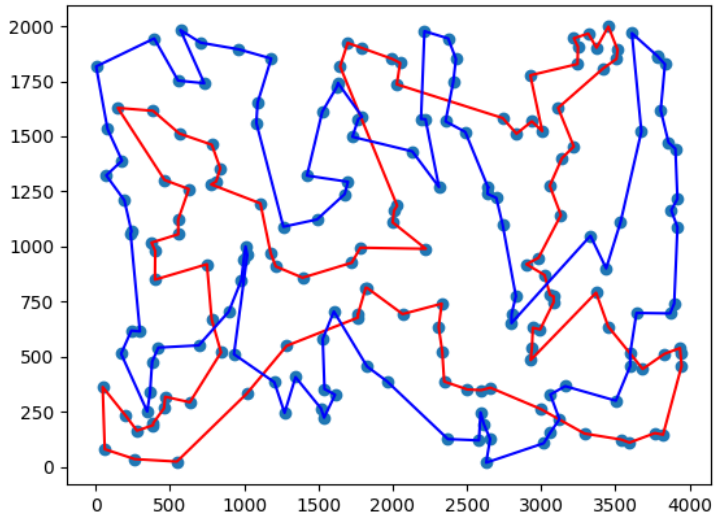


Rysunek 9: Rozwiązanie dla Random_walk na instancji kroA200

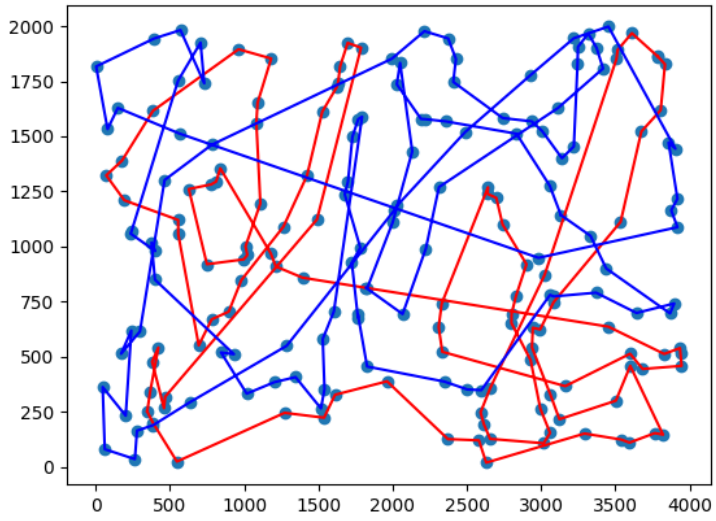
6.2 Instancja kroB200



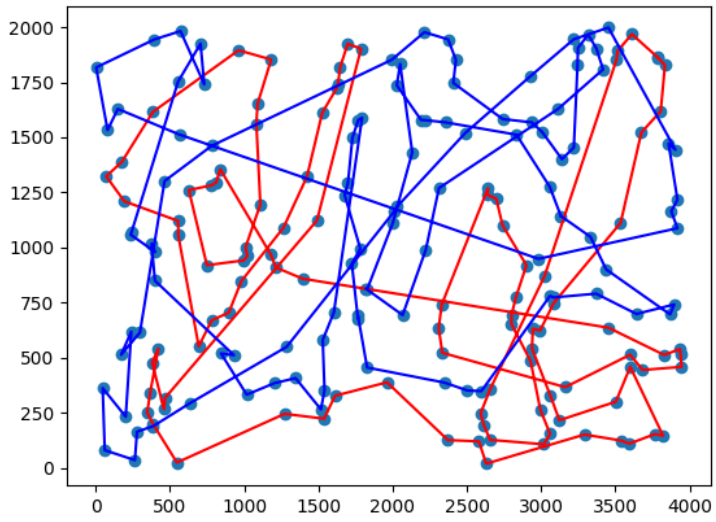
Rysunek 10: Rozwiązanie dla Steepest_vertices_random na instancji kroB200



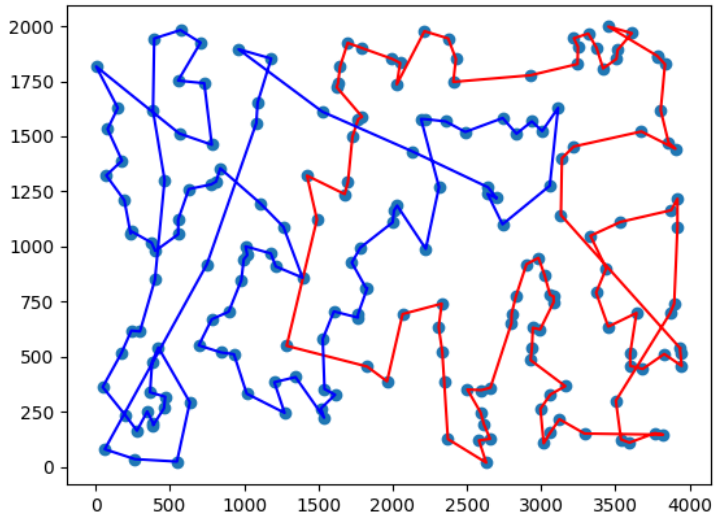
Rysunek 11: Rozwiązanie dla Steepest_edge_random na instancji kroB200



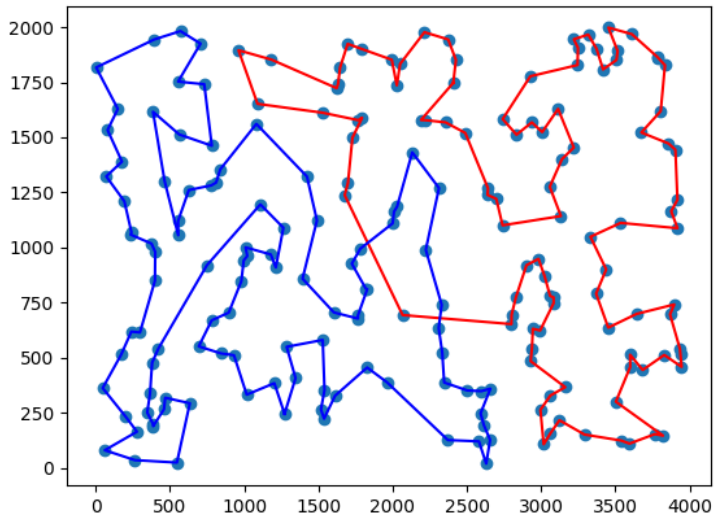
Rysunek 12: Rozwiązanie dla Greedy_vertices_random na instancji kroB200



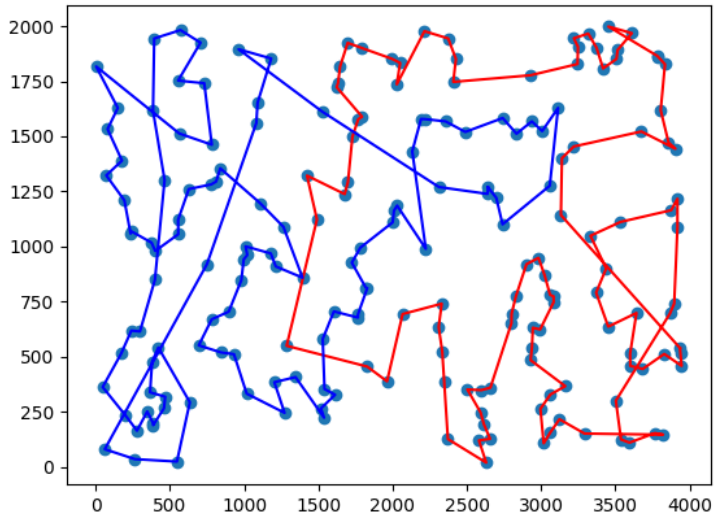
Rysunek 13: Rozwiązanie dla Greedy_edge_random na instancji kroB200



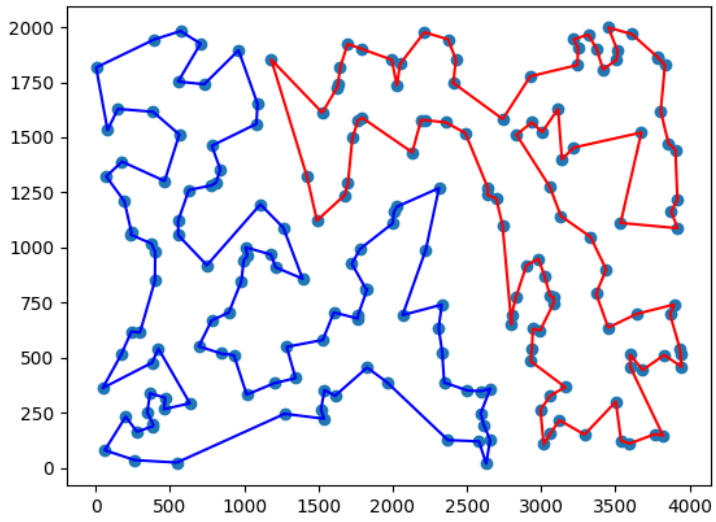
Rysunek 14: Rozwiązanie dla Steepest_vertices_greedy na instancji kroB200



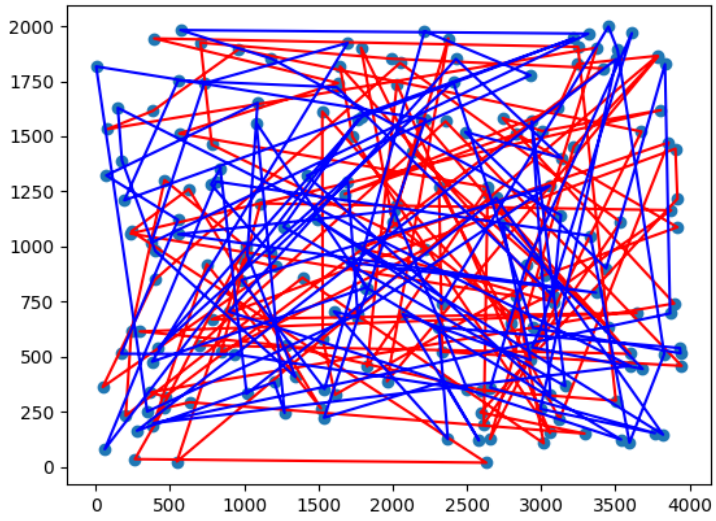
Rysunek 15: Rozwiązanie dla Steepest_edge_greedy na instancji kroB200



Rysunek 16: Rozwiązanie dla Greedy_vertices_greedy na instancji kroB200



Rysunek 17: Rozwiązanie dla Greedy_edge_greedy na instancji kroB200



Rysunek 18: Rozwiązanie dla Random_walk na instancji kroB200

7 Wnioski

Na podstawie przeprowadzonych eksperymentów można wyciągnąć następujące wnioski:

1. Najlepsze wyniki dla obu instancji osiągnęła kombinacja Steepest_edges_greedy czyli lokalne przeszukiwanie strome wykorzystujące zamianę krawędzi oraz posiadające rozwiązanie początkowe wykonane metodą zachłanną, uzyskując najniższe wartości funkcji celu (34214 dla kroa200 i 34752 dla krob200).

2. Metoda Steepest_vertices_greedy jest najszybsza (średni czas wykonania poniżej 0.5 sekundy), jednak jej wyniki są gorsze pod względem jakości rozwiązania (39135 dla kroa200 i 39302 dla krob200) w porównaniu do najlepszych metod opartych na krawędziach.

3. Metody oparte na strategii początkowym rozwiązaniu greedy oferują dobry kompromis między jakością rozwiązania a czasem wykonania, osiągając lepsze wyniki niż metody z losowym rozwiązaniem startowym przy znacznym skróceniu czasu obliczeń.

4. Metody wykorzystujące losowe rozwiązanie początkowe i zamianę wierzchołków są bardziej czasochłonne niż te bazujące na podejściu greedy, a

ich wyniki są znacznie gorsze. Na przykład metoda Greedy_vertices_random uzyskała wynik 66869 dla kroa200 przy czasie 17.62 s.

5. Metoda losowego błądzenia jest najmniej efektywna zarówno pod względem jakości rozwiązania, jak i czasu wykonania. Uzyskała najgorsze wartości funkcji celu (281003 dla kroa200 i 276016 dla krob200) przy czasie działania równym czasowi najwolniejszego algorymu przeszukiwania.

6. Wyniki dla instancji kroa200 i krob200 są spójne, co sugeruje, że rozwiązania poszczególnych metod nie zależą od konkretnej instancji problemu.

7. W celu uzyskania najlepszej jakości rozwiązania powinno zastosować się metodę Steepest_edges_greedy lub Greedy_edges_greedy, natomiast jeśli priorytetem jest szybkość obliczeń,najlepszym wyborem wyborem będzie Steepest_vertices_greedy.

8. Najlepsze rezultaty można osiągnąć, łącząc podejście zachłanne do znalezienia rozwiązania początkowego z dalszą optymalizacją poprzez zamianę krawędzi. Strategia ta pozwala szybko uzyskać dobrą jakość rozwiązania, a następnie ulepszać je poprzez lokalne modyfikacje.

8 Kod źródłowy

Pełny kod źródłowy implementacji wszystkich algorytmów jest dostępny w repozytorium GitHub: `https://github.com/FilipRosiak1/IMO-2`