

# IMO – LAB 5 Hybrydowy Algorytm Ewolucyjny

Jan Bróździak 141142

## Opis zadania

W ramach niniejszego zadania implementowany jest Hybrydowy Algorytm Ewolucyjny (HAE), który łączy mechanizmy algorytmów ewolucyjnych z lokalnym przeszukiwaniem, w celu znalezienia lepszych rozwiązań dla problemów optymalizacji kombinatorycznej. Celem eksperymentu jest porównanie efektywności HAE z wcześniej zaimplementowanymi metodami, takimi jak Multiple Start Local Search (MSLS) oraz Iterated Local Search (ILS), co pozwoli na ocenę skuteczności hybrydyzacji technik ewolucyjnych i lokalnego przeszukiwania.

## Opis algorytmów w pseudokodzie

### Inicjalizacja populacji

```
Function initialize_population(population_size, distance_matrix)
    Initialize empty population list
    Initialize empty solutions_set to track unique scores

    While population size is less than population_size
        Generate random cycles (cycle1, cycle2) from distance_matrix
        Perform local search on the random cycles
        (cycle1, cycle2, score) =
steepst_ascent_local_search(cycle1, cycle2, distance_matrix,
generate_combined_moves_list_2)

        If score not in solutions_set
            Add (cycle1, cycle2, score) to population
            Add score to solutions_set
        End If
    End While

    Return population
End Function
```

## Naprawa rozwiązania

```
Function repair_solution(cycle1, cycle2, remaining_vertices,  
distance_matrix)  
    Initialize visited set with vertices in cycle1 and cycle2  
    While remaining_vertices is not empty  
        Insert a vertex into cycle1 using 2-regret heuristic  
        Update remaining_vertices  
        If remaining_vertices is not empty  
            Insert a vertex into cycle2 using 2-regret heuristic  
            Update remaining_vertices  
    End While  
  
    Return cycle1, cycle2  
End Function
```

## Rekombinacja

```
Function recombination(parent1, parent2, distance_matrix)
    Extract cycle1 and cycle2 from parent1 and parent2

    Initialize offspring_cycle1 and offspring_cycle2 as empty lists

    For each edge in cycle1 of parent1
        If the edge exists in parent2, add the vertex to
offspring_cycle1
    End For

    For each edge in cycle2 of parent1
        If the edge exists in parent2, add the vertex to
offspring_cycle2
    End For

    Remove free-floating vertices from offspring_cycle1 and
offspring_cycle2

    Determine remaining vertices not in the offspring cycles

    Repair offspring cycles using the regret heuristic
    (repaired_cycle1, repaired_cycle2) =
repair_solution(offspring_cycle1, offspring_cycle2,
remaining_vertices, distance_matrix)

    Return repaired_cycle1, repaired_cycle2
End Function
```

## Steady state

```
Function steady_state_selection(population, distance_matrix,
max_iterations, time_limit)
    Set population_size to the length of population
    Set iteration to 0
    Record start_time

    While iteration < max_iterations and time has not exceeded
time_limit
        Select two parents randomly from the population

        Generate offspring by recombination of parent1 and parent2
        (offspring_cycle1, offspring_cycle2) = recombination(parent1,
parent2, distance_matrix)

        Perform local search on the offspring
        (offspring_cycle1, offspring_cycle2, offspring_score) =
steepest_ascent_local_search(
            offspring_cycle1, offspring_cycle2, distance_matrix,
generate_combined_moves_list_2
        )

        If offspring_score is not already in the population
            Find the worst solution in the population
            If offspring_score is better than the worst solution
                Replace the worst solution with the offspring
            End If
        End If

        Increment iteration
    End While

    Return the best solution from the population
```

## HAE

```
Function hybrid_evolutionary_algorithm(distance_matrix,  
population_size, max_iterations, time_limit)  
    Initialize the population  
    population = initialize_population(population_size,  
distance_matrix)  
  
    Perform steady-state selection to evolve the population  
    best_solution = steady_state_selection(population,  
distance_matrix, max_iterations, time_limit)  
  
    Return best_solution  
End Function
```

## Parametry uruchomienia HAE

**Populacja elitarna:** 20

**Max limit czasu:** 600 (potem zmniejszony do 60)

**Max ilość iteracji wewnętrznych:** 3000 (potem zmniejszona do 1000)

## Wizualizacje, wnioski, przemyślenia

Byłem bardzo pozytywnie zaskoczony, jak szybko algorytm działa. Początkowo, nauczony poprzednimi metodami, ustawiłem limit czasu na 600 sekund, i ilość iteracji na 3000, jednak algorytm zbiegał na tyle szybko, że można było spokojnie zmniejszyć te wartości. Poniżej wizualizacja zbiegania dla instancji KRO200B, oraz znalezione najlepsze rozwiązanie

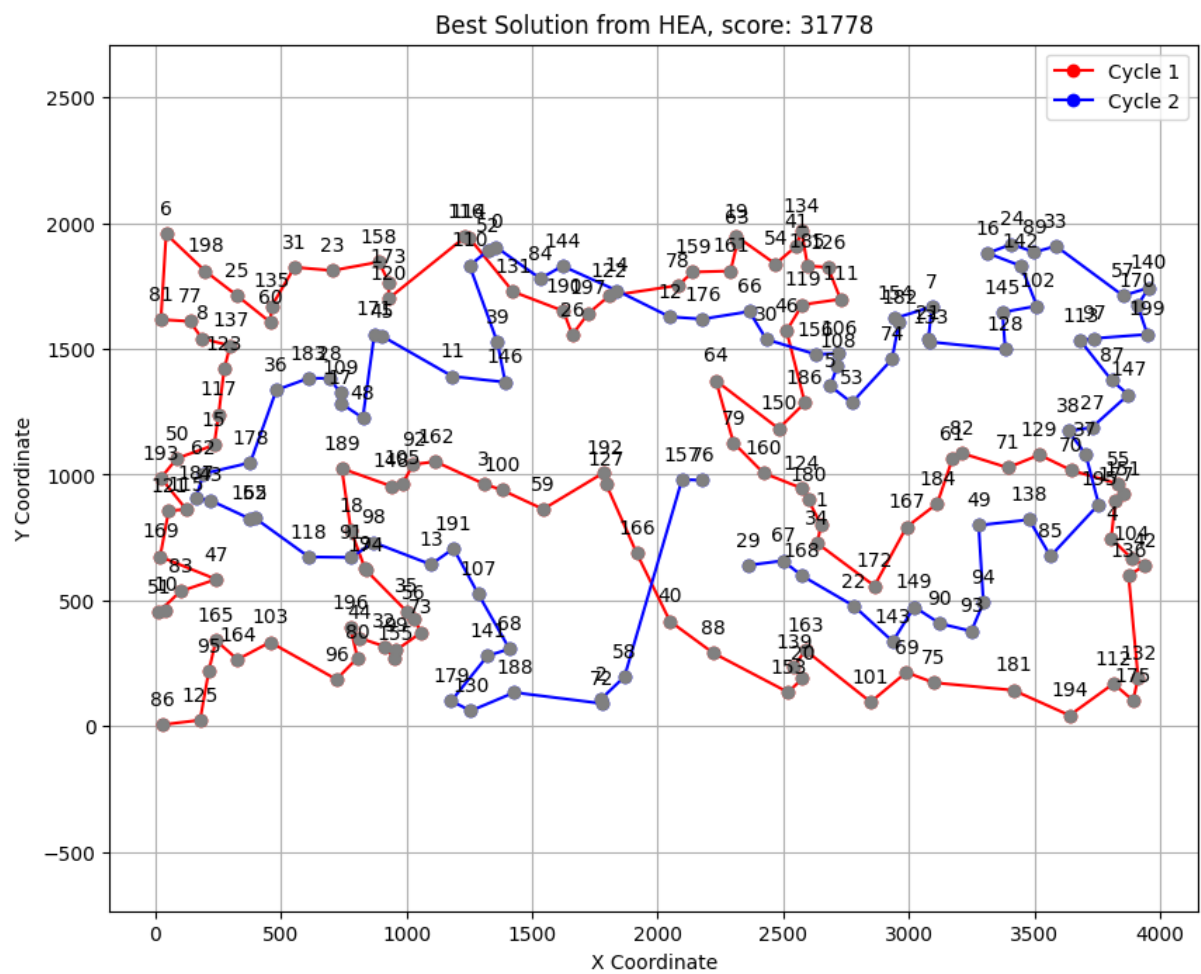


Poniżej zaprezentowane jest porównanie HAE z metodami MSLS oraz ILS1 i ILS2. HAE był uruchamiany 10 razy dla każdej instancji:

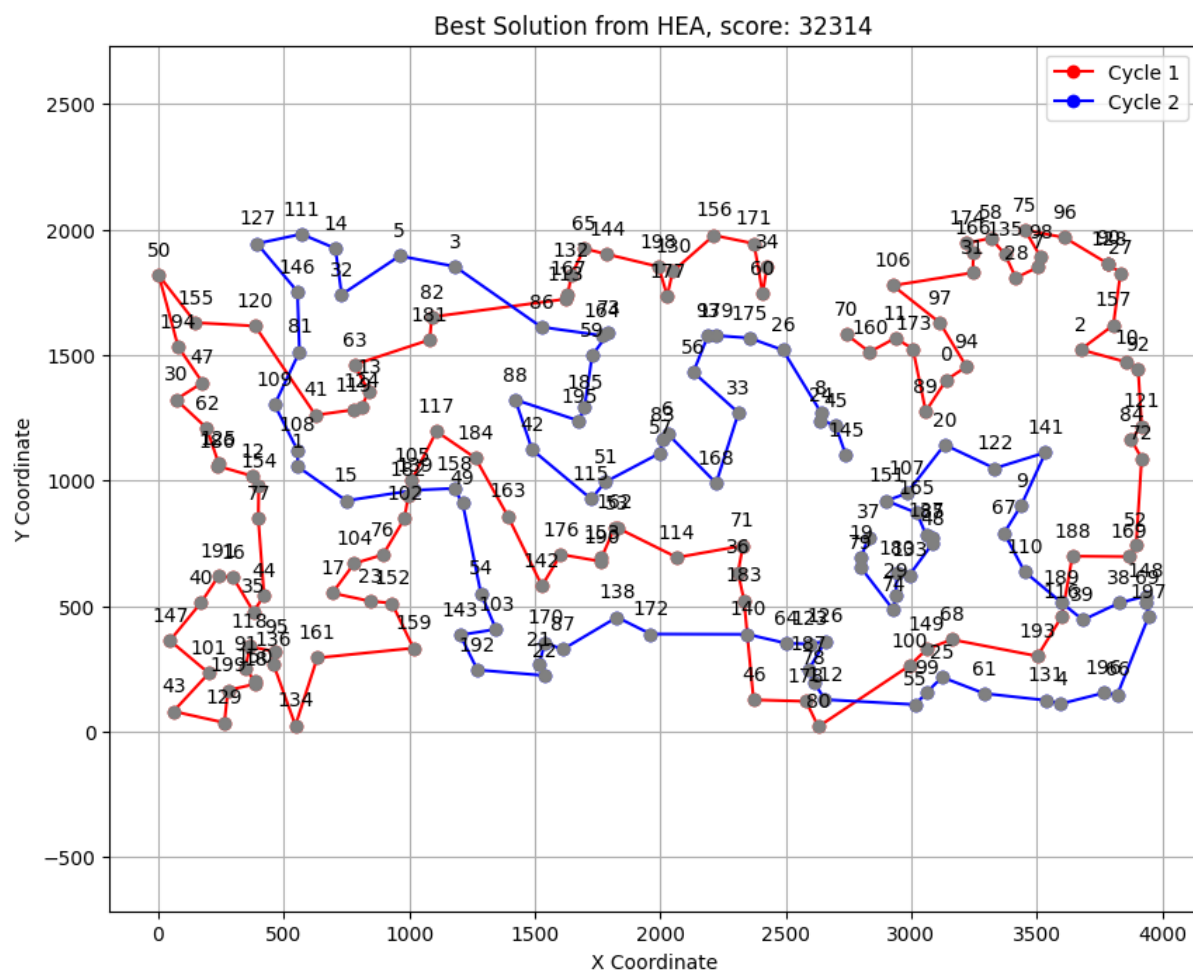
Metoda	Instancja	Min	Max	Avg
MSLS	KRO200A	35341	36510	35945
ILS1	KRO200A	33345	36223	34409
ILS2	KRO200A	31913	34204	32782
HAE	KRO200A	31778	33661	32784
MSLS	KRO200B	35638	36853	36161
ILS1	KRO200B	34447	36264	35438
ILS2	KRO200B	32920	34841	33752
HAE	KRO200B	32314	34044	33344

Algorytm HAE dla instancji KRO200A uzyskał średnio niemalże identyczne wyniki jak ILS2. Dla instancji KRO200B wyniki są nieznacznie lepsze. Warto w tym miejscu odnotować, że HAE osiągał takie wyniki w dużo krótszym czasie.

## Wizualizacja najlepszych rozwiązań dla HAE



Rysunek 1 KRO200A



Rysunek 2 KRO200B