Komivoyager

May 21, 2024

1 Algorytmy genetyczne - problem Komiwojażera

1.0.1 Import bibliotek

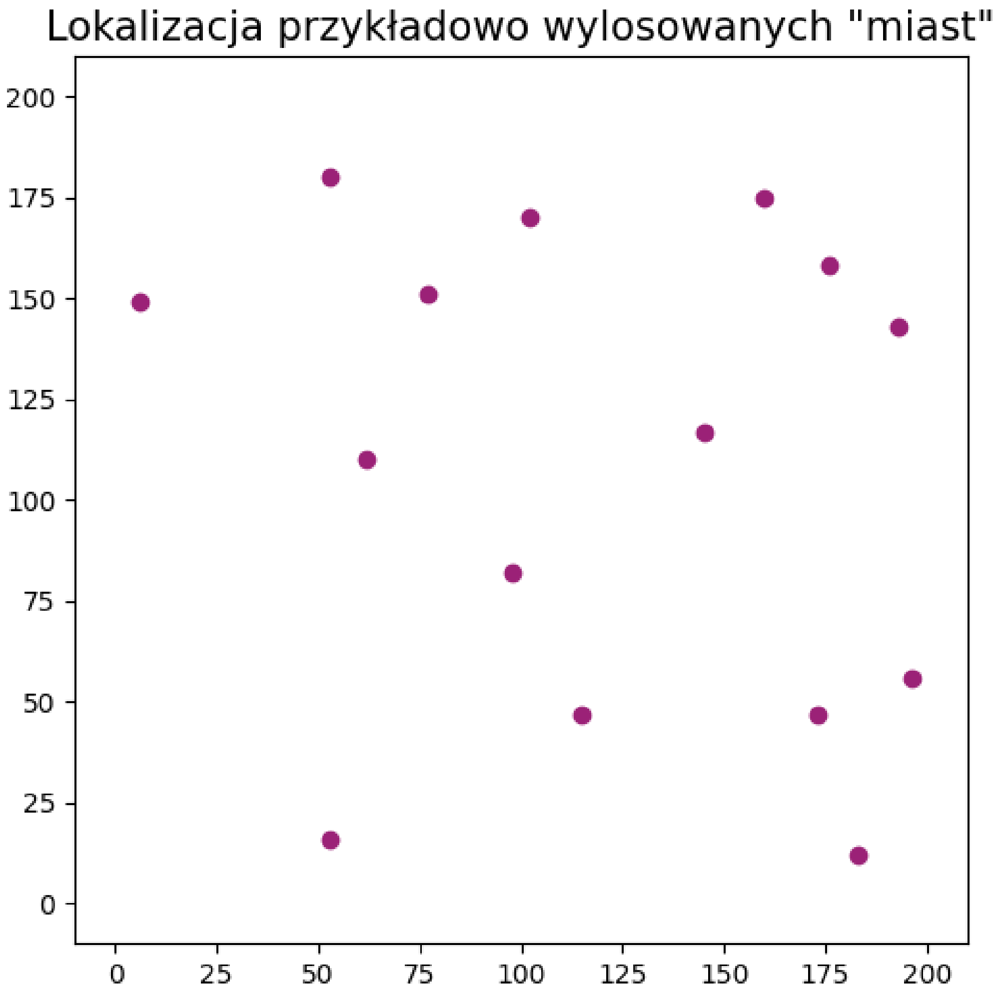
[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import random
from scipy.spatial import distance_matrix
```

1.0.2 Funkcja generująca punkty

[2]:
```
def generate_cities(n_cities, area_size):
    np.random.seed(26)
    return pd.DataFrame({"X": np.random.randint(area_size, size=n_cities), "Y":␣
 ↪np.random.randint(area_size, size=n_cities)})
```

[3]:
```
cities = generate_cities(15, 200)
plt.figure(figsize = (6,6))
plt.scatter(cities.X, cities.Y, color="#9B2177")
plt.title('Lokalizacja przykładowo wylosowanych "miast"', fontsize=15)
plt.xlim(-10,210)
plt.ylim(-10,210)
```

[3]:
```
(-10.0, 210.0)
```



1.0.3 Przedstawienie koordynatów

[4]:
```
display(cities.T)
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
X	53	62	6	176	193	102	77	160	196	98	173	145	183	115	53
Y	180	110	149	158	143	170	151	175	56	82	47	117	12	47	16

1.0.4 Macierz dystansu

[5]:
```
def distances(cities):
    return pd.DataFrame(distance_matrix(cities.values, cities.values),␣
 ↪index=cities.index, columns=cities.index)
```

1.0.5 Wybór populacji początkowej

[6]:
```
def genesis(cities, num_individuals):
    n_cities = len(cities)
    population_set = []
    for i in range(num_individuals):
        city = np.random.permutation(list(range(1, n_cities)))
        population_set.append(city)
    population_df = pd.DataFrame(population_set, columns=['City_' + str(i) for␣
 ↪i in range(1, n_cities)])
    return population_df
```

1.0.6 Ocena populacjii

[7]:
```
def compute_fitness(population_df, distance_matrix):
    fitness_values = []
    for idx, solution in population_df.iterrows():
        solution_distance = 0
        for i in range(len(solution) - 1):
            city1 = solution.iloc[i]
            city2 = solution.iloc[i + 1]
            solution_distance += distance_matrix.loc[city1, city2]
        first_city = solution.iloc[0]
        last_city = solution.iloc[len(solution) - 1]
        solution_distance += distance_matrix.loc[0, first_city]
        solution_distance += distance_matrix.loc[last_city, 0]
        fitness_values.append(solution_distance)
    return fitness_values
```

1.0.7 Selekcja - metoda ruletki

[8]:
```
def roulette_selection(fitness_list, population_df):
    inverse_fitness_list = 1 / np.array(fitness_list)
    total_fit = inverse_fitness_list.sum()
    prob_list = inverse_fitness_list/total_fit
    parents_indices = np.random.choice(list(range(len(population_df))), size=2,␣
 ↪p=prob_list, replace=False)
    return population_df.loc[parents_indices[0],:], population_df.
 ↪loc[parents_indices[1],:]
```

1.0.8 Selekcja - metoda rankingowa

[9]:
```
def ranking_selection(fitness_list, population_df):
    min_indices = np.argsort(fitness_list)[:5]
    selected_population = population_df.iloc[min_indices].copy()
    selected_parents_indices = np.random.choice(len(selected_population),␣
 ↪size=2, replace=False)
```

```
    selected_parents = selected_population.iloc[selected_parents_indices]

    return selected_population.iloc[0], selected_population.iloc[1]
```

1.0.9 Krzyżowanie

[10]:
```
def fix_duplicates(child, parent1, parent2):
    seen = set()
    duplicates = []
    for index, gene in enumerate(child):
        if gene in seen:
            duplicates.append(index)
        else:
            seen.add(gene)
    missing = set(parent1) - seen
    for index in duplicates:
        child.iloc[index] = missing.pop()
    return child
```

[11]:
```
def crossover(parent_a, parent_b, crossover_prob):
    n_cities = len(parent_a)
    if np.random.rand() < crossover_prob:
        start_point = np.random.randint(1, n_cities - 1)
        end_point = np.random.randint(start_point + 1, n_cities)
        child_a = pd.concat([parent_a[:start_point],parent_b[start_point:
↪end_point],parent_a[end_point:]])
        child_b = pd.concat([parent_b[:start_point],parent_a[start_point:
↪end_point],parent_b[end_point:]])
        child_a = fix_duplicates(child_a, parent_a, parent_b)
        child_b = fix_duplicates(child_b, parent_a, parent_b)
    else:
        child_a, child_b = parent_a, parent_b
    return child_a, child_b
```

1.0.10 Mutacja

[12]:
```
def mutate(individ, mutation_prob = 0.8):
    if np.random.rand() < mutation_prob:
        n_cities = len(individ)
        idx1, idx2 = np.random.choice(n_cities, 2, replace=False)
        individ.iloc[idx1], individ.iloc[idx2] = individ.iloc[idx2], individ.
↪iloc[idx1]
    return individ
```

1.0.11 Generowanie nowej populacji

[13]:
```
def create_new_generation(current_population_df, fitness_values,␣
↪num_individuals, crossover_prob, mutation_prob, selection_type):
    new_generation = []

    while len(new_generation) < num_individuals:

        if selection_type == 'roulette':
            parent_1, parent_2 = roulette_selection(fitness_values,␣
↪current_population_df)
        elif selection_type == 'ranking':
            parent_1, parent_2 = ranking_selection(fitness_values,␣
↪current_population_df)

        child_1, child_2 = crossover(parent_1, parent_2, crossover_prob)
        child_1 = mutate(child_1, mutation_prob)
        child_2 = mutate(child_2, mutation_prob)
        new_generation.extend([child_1, child_2])

    new_generation_df = pd.DataFrame(new_generation,␣
↪columns=current_population_df.columns).reset_index(drop=True)
    return new_generation_df
```

1.1 Implementacja algorytmu genetycznego

[14]:
```
def genetic_algorithm(num_cities, num_individuals, generations, mutation_rate=0.
↪02, crossover_rate=0.8, area_size=200, selection_type='roulette'):
    cities = generate_cities(num_cities, area_size)
    dist = distances(cities)
    population = genesis(cities.index, num_individuals)

    best_fitness_over_generations = []
    avg_fitness_over_generations = []
    median_fitness_over_generations = []
    all_fitness_scores = []

    for generation in range(generations):
        fitness = compute_fitness(population, dist)
        best_fitness_over_generations.append(min(fitness))
        avg_fitness_over_generations.append(np.mean(fitness))
        median_fitness_over_generations.append(np.median(fitness))
        all_fitness_scores.append(fitness)
        population = create_new_generation(population, fitness,␣
↪num_individuals, crossover_rate, mutation_rate, selection_type)

    best_idx = np.argmin(fitness)
```

```
    best_individual = population.iloc[best_idx, :].values
    best_distance = fitness[best_idx]

    return best_individual, best_distance, all_fitness_scores,␣
↪best_fitness_over_generations, avg_fitness_over_generations,␣
↪median_fitness_over_generations, cities
```

1.1.1 Najlepsza trasa w danym pokoleniu

Funkcja do wizualizacji

[15]:
```
def best_route(best_individual, cities_df):
    x = [cities.X.loc[i] for i in best_individual]
    y = [cities.Y.loc[i] for i in best_individual]

    x.insert(0,cities.X.loc[0])
    y.insert(0,cities.Y.loc[0])

    x.append(cities.X.loc[0])
    y.append(cities.Y.loc[0])

    plt.figure(figsize=(6, 6))
    plt.plot(x,y, color="#efaffa")
    plt.scatter(cities_df["X"], cities_df["Y"], color="#8f00a8")
    plt.scatter(cities_df["X"].loc[0], cities_df["Y"].loc[0], color="#419bf8",␣
↪s=200, marker="*")
    plt.xlim(0, 210)
    plt.ylim(0, 210)
    plt.title("Najlepsza wyznaczona trasa w generacji", fontsize=16)
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.grid(True)
```

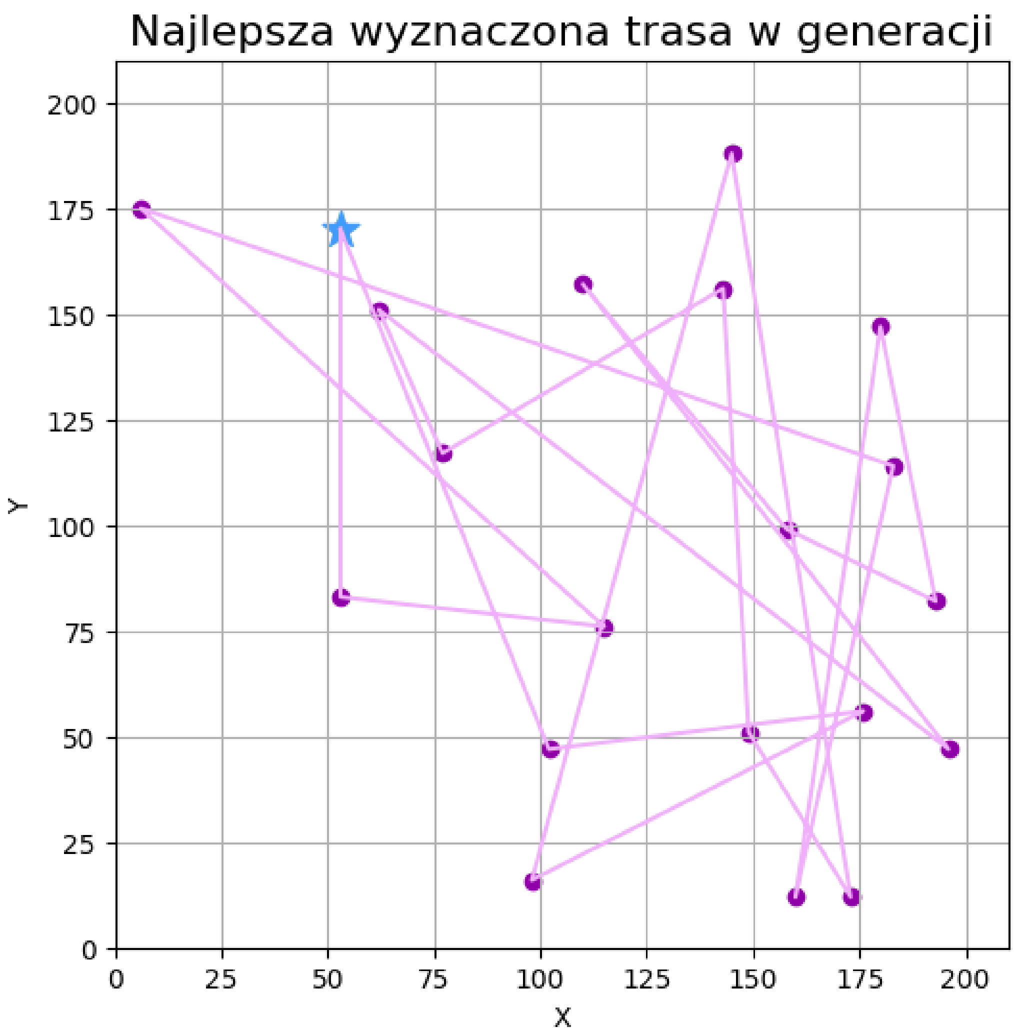
1.1.2 Funkcja wizualizująca poprawę jakości

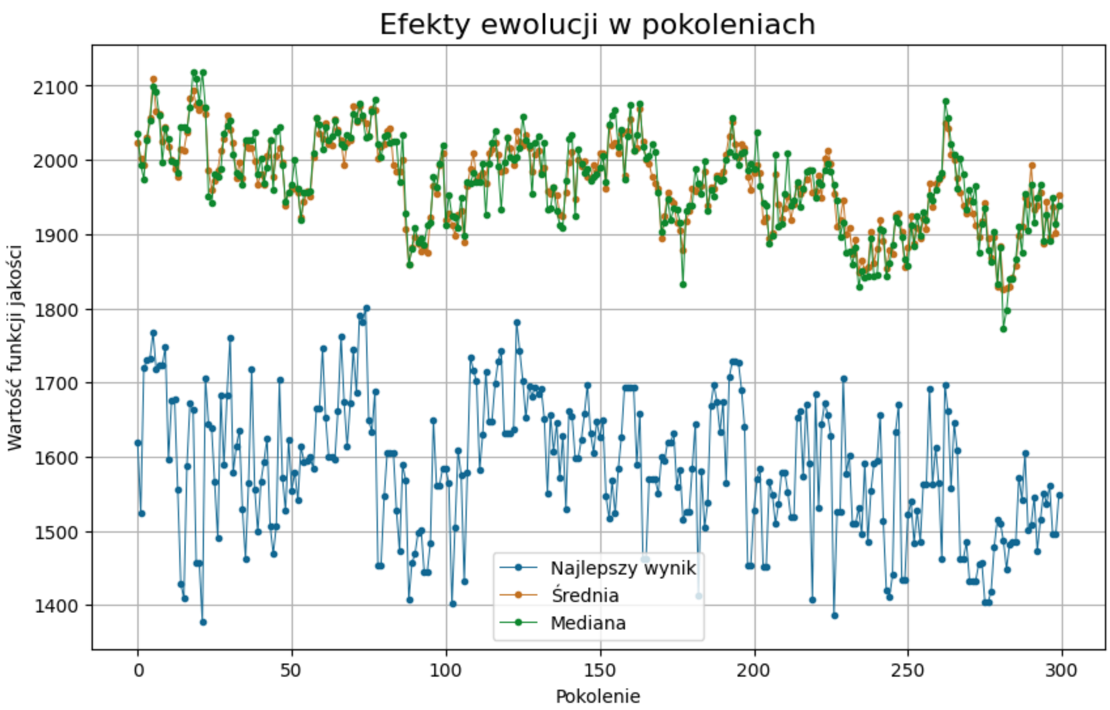
[16]:
```
def plot_evolution(best_fitness, avg_fitness, median_fitness, generations):
    plt.figure(figsize=(10, 6))
    x=range(generations)
    plt.plot(x, best_fitness, 'o-', label='Najlepszy wynik', color="#106691",␣
↪linewidth=0.7, markersize=3)
    plt.plot(x, avg_fitness, 'o-', label='Średnia', color="#c2701f",␣
↪linewidth=0.7, markersize=3)
    plt.plot(x, median_fitness, 'o-', label='Mediana', color="#0e872e",␣
↪linewidth=0.7, markersize=3)
    plt.xlabel('Pokolenie')
    plt.ylabel('Wartość funkcji jakości')
    plt.title('Efekty ewolucji w pokoleniach', fontsize=16)
    plt.legend()
```

```
    plt.grid(True)
    plt.show()
```

1.1.3 Prawdopodobieństwo mutacji 0.4, krzyżowanie 90%

[17]:
```
best_individual, best_distance, all_fitness_scores,␣
 ↪best_fitness_over_generations, avg_fitness_over_generations,␣
 ↪median_fitness_over_generations, cities = genetic_algorithm(num_cities=20,␣
 ↪num_individuals=50, generations=300, mutation_rate=0.4, crossover_rate=0.9)
best_route(best_individual, cities)
plot_evolution(best_fitness_over_generations, avg_fitness_over_generations,␣
 ↪median_fitness_over_generations, generations=300)
print(f'Najlepsza znaleziona trasa ma długość: {best_distance:.3f}')
```

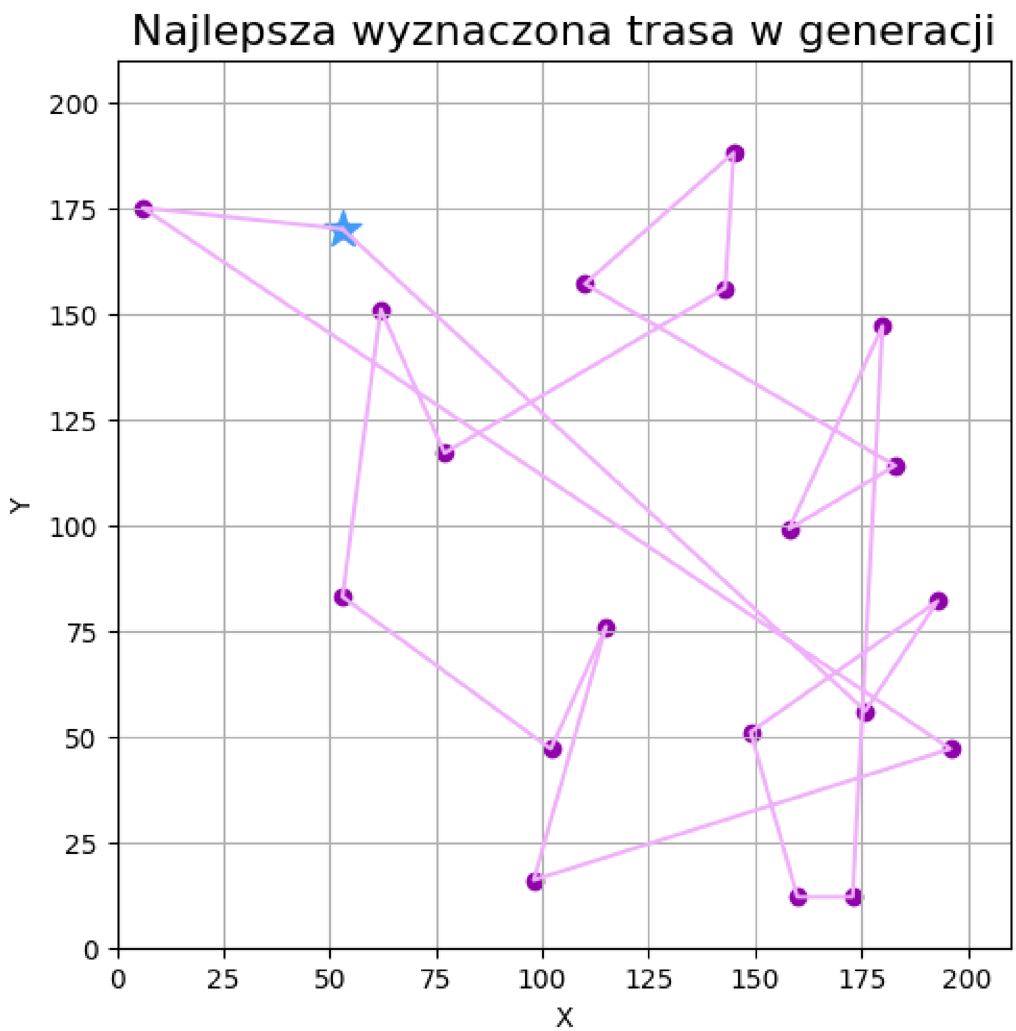


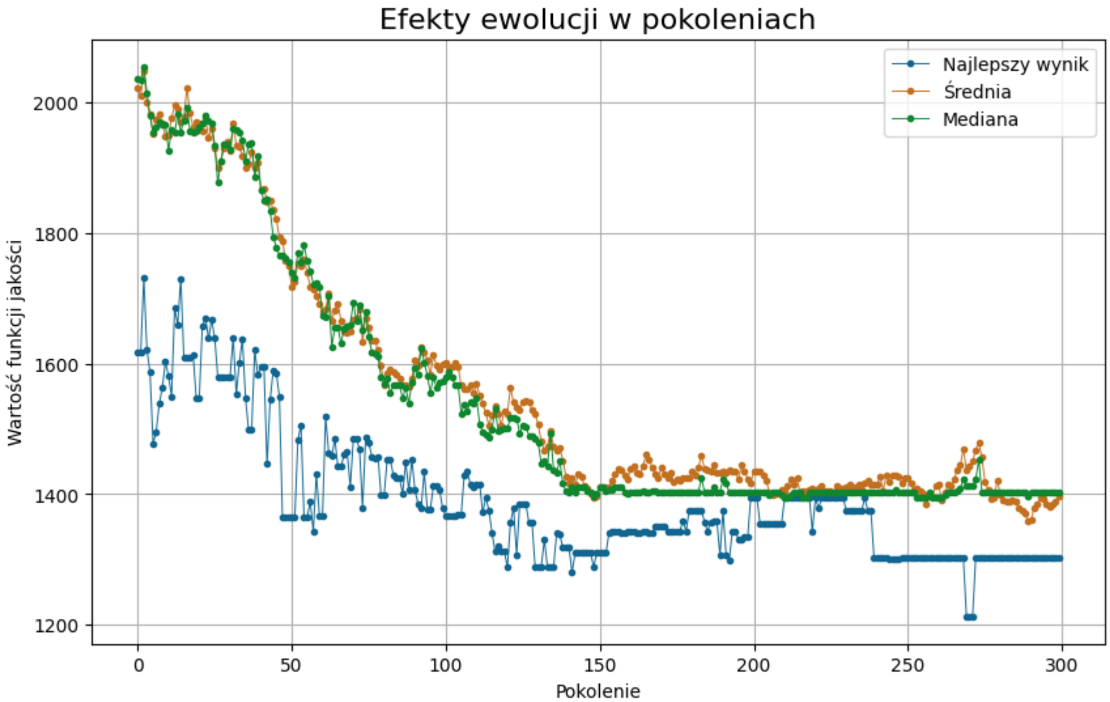



```
Najlepsza znaleziona trasa ma długość: 1548.541
```

1.1.4 Prawdopodobieństwo mutacji 0.02, krzyżowanie 90%

[18]:
```
best_individual, best_distance, all_fitness_scores,␣
↪best_fitness_over_generations, avg_fitness_over_generations,␣
↪median_fitness_over_generations, cities = genetic_algorithm(num_cities=20,␣
↪num_individuals=50, generations=300, mutation_rate=0.02, crossover_rate=0.9)
best_route(best_individual, cities)
plot_evolution(best_fitness_over_generations, avg_fitness_over_generations,␣
↪median_fitness_over_generations, generations=300)
print(f'Najlepsza znaleziona trasa ma długość: {best_distance:.3f}')
```

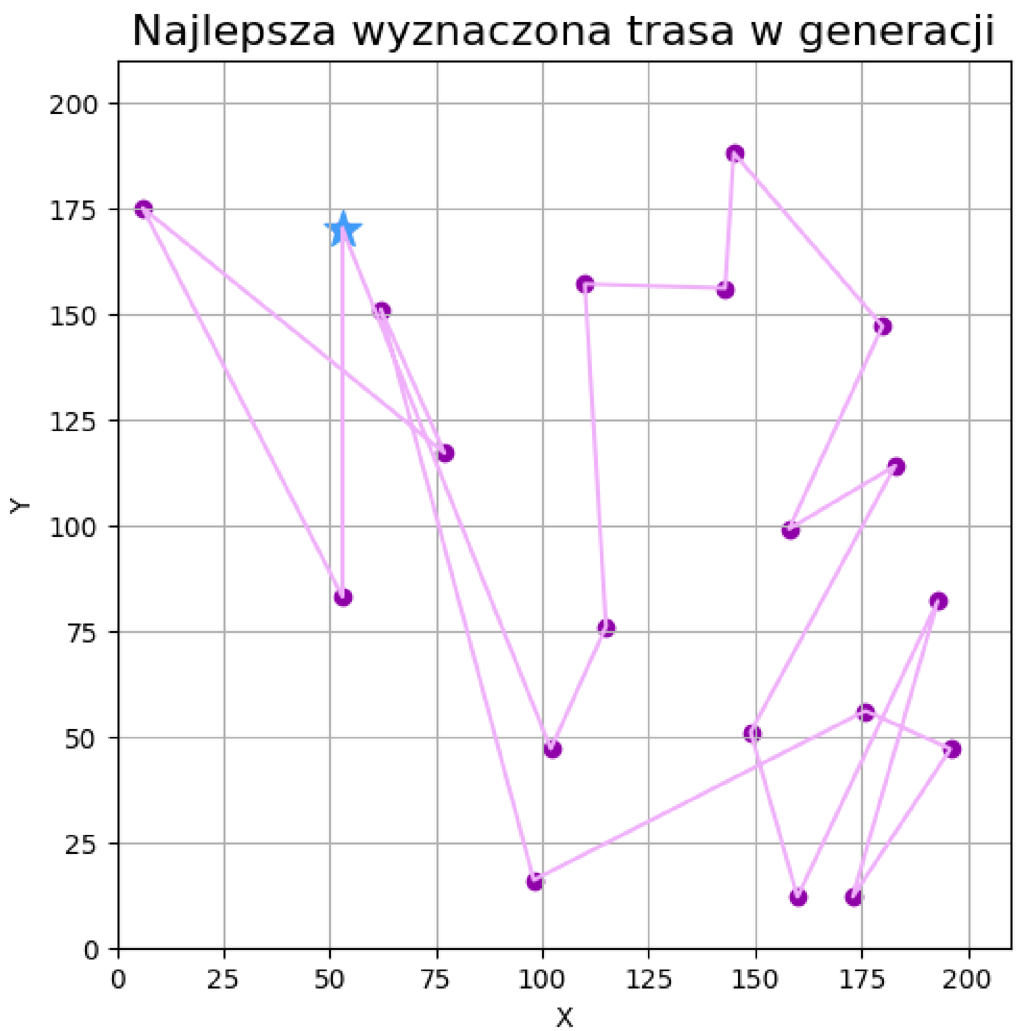

Najlepsza wyznaczona trasa w generacji
200
175
150
125
100
75
50
25
0
Y
0
25
50
75
100
125
150
175
200
X

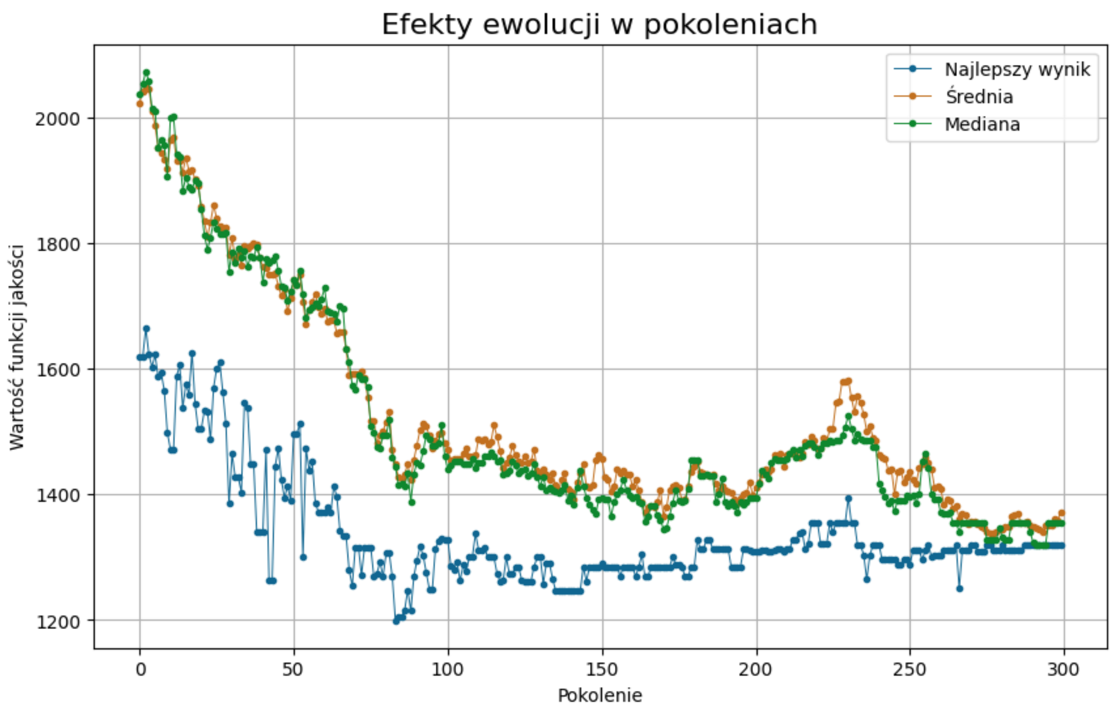


Najlepsza znaleziona trasa ma długość: 1302.618

1.1.5 Prawdopodobieństwo mutacji 0.02, krzyżowanie 95%

[19]:
```
best_individual, best_distance, all_fitness_scores,␣
↪best_fitness_over_generations, avg_fitness_over_generations,␣
↪median_fitness_over_generations, cities = genetic_algorithm(num_cities=20,␣
↪num_individuals=50, generations=300, mutation_rate=0.02, crossover_rate=0.95)
best_route(best_individual, cities)
plot_evolution(best_fitness_over_generations, avg_fitness_over_generations,␣
↪median_fitness_over_generations, generations=300)
print(f'Najlepsza znaleziona trasa ma długość: {best_distance:.3f}')
```

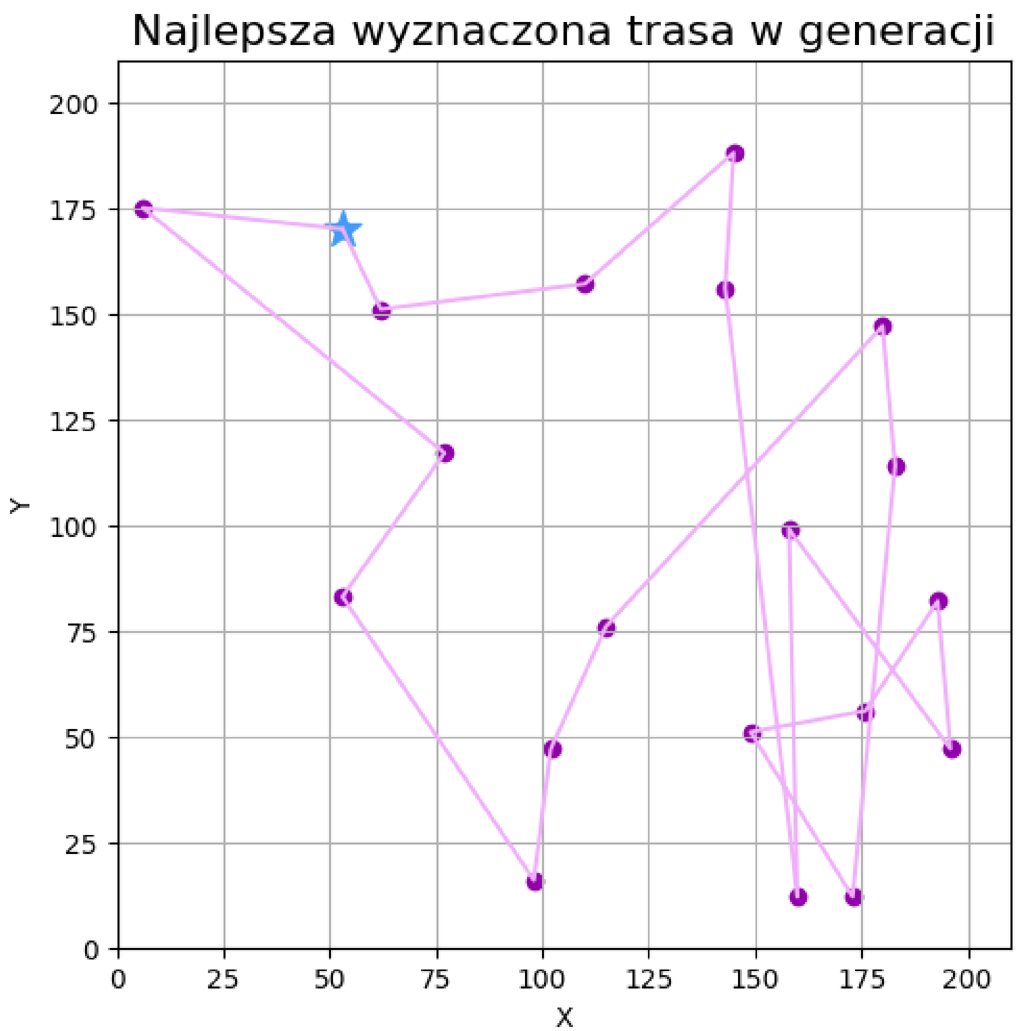

Najlepsza wyznaczona trasa w generacji
200
175
150
125
100
75
50
25
0
Y
0
25
50
75
100
125
150
175
200
X

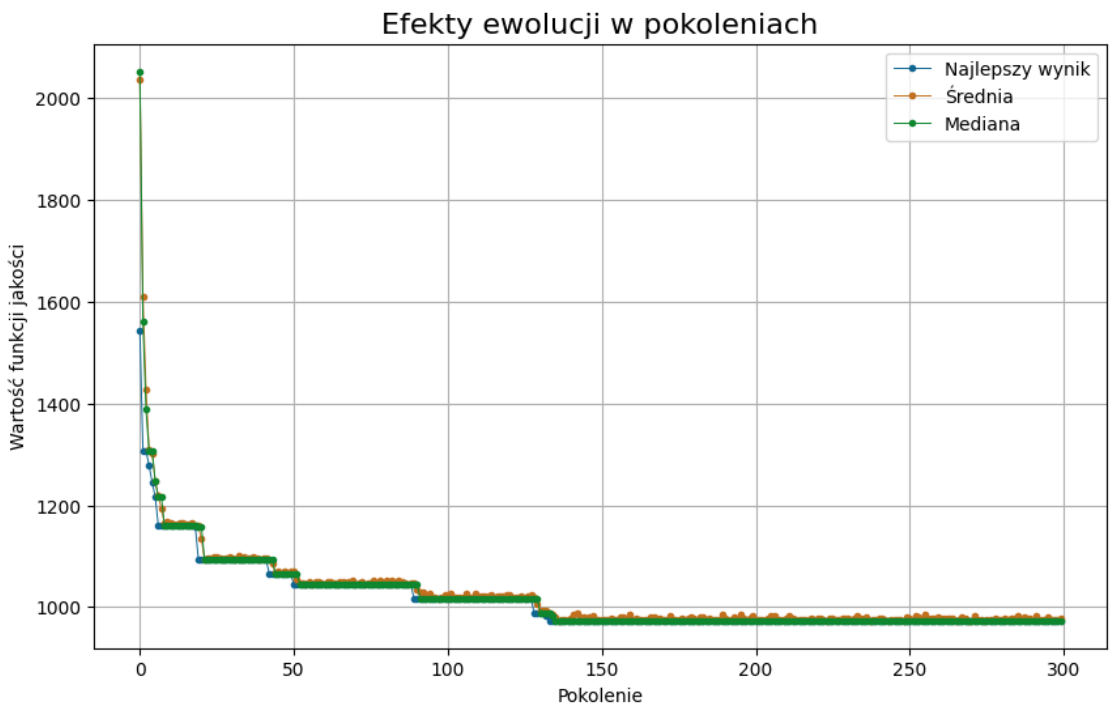



```
Najlepsza znaleziona trasa ma długość: 1318.917
```

1.1.6 Użycie innej metody selekcji

[20]:
```python
best_individual, best_distance, all_fitness_scores,␣
↪best_fitness_over_generations, avg_fitness_over_generations,␣
↪median_fitness_over_generations, cities = genetic_algorithm(num_cities=20,␣
↪num_individuals=150, generations=300, mutation_rate=0.02, crossover_rate=0.
↪95, selection_type='ranking')
best_route(best_individual, cities)
plot_evolution(best_fitness_over_generations, avg_fitness_over_generations,␣
↪median_fitness_over_generations, generations=300)
print(f'Najlepsza znaleziona trasa ma długość: {best_distance:.3f}')
```

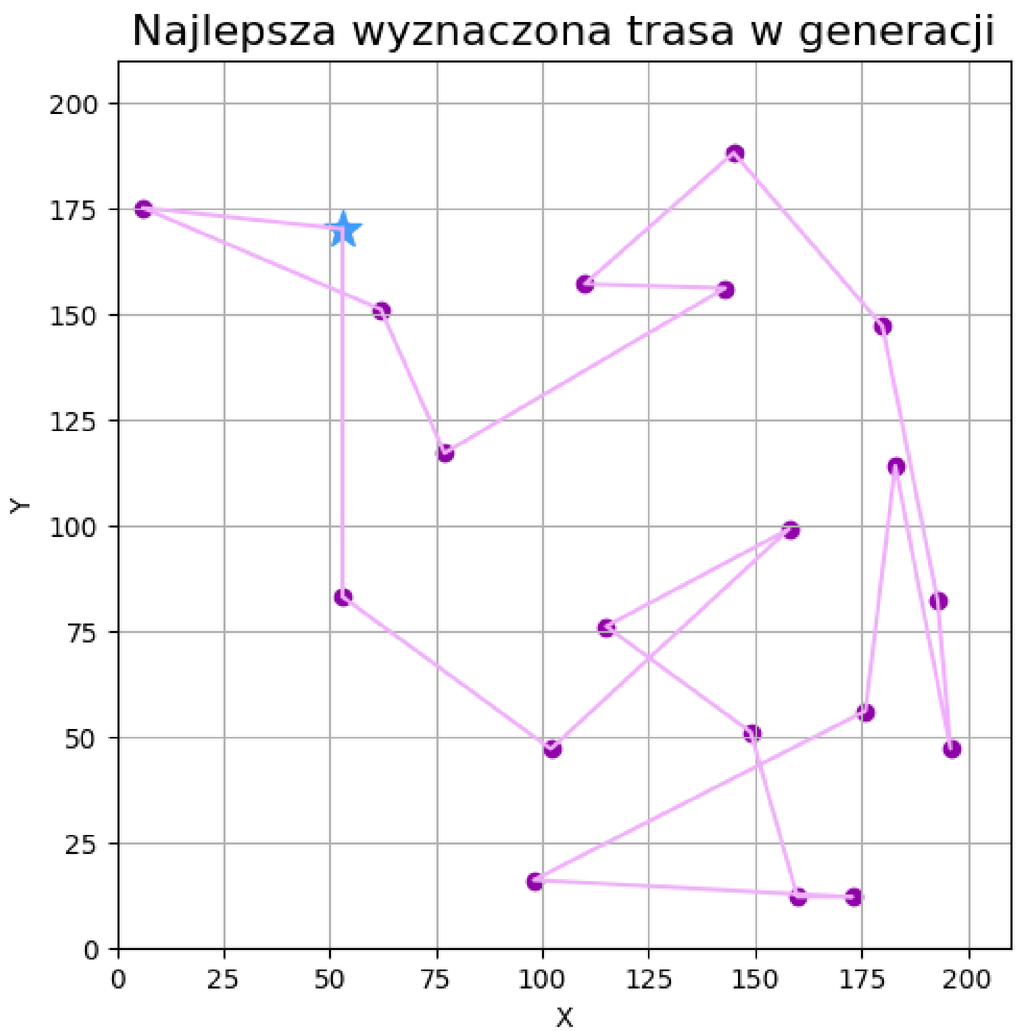

Najlepsza wyznaczona trasa w generacji
Y
X
0
25
50
75
100
125
150
175
200

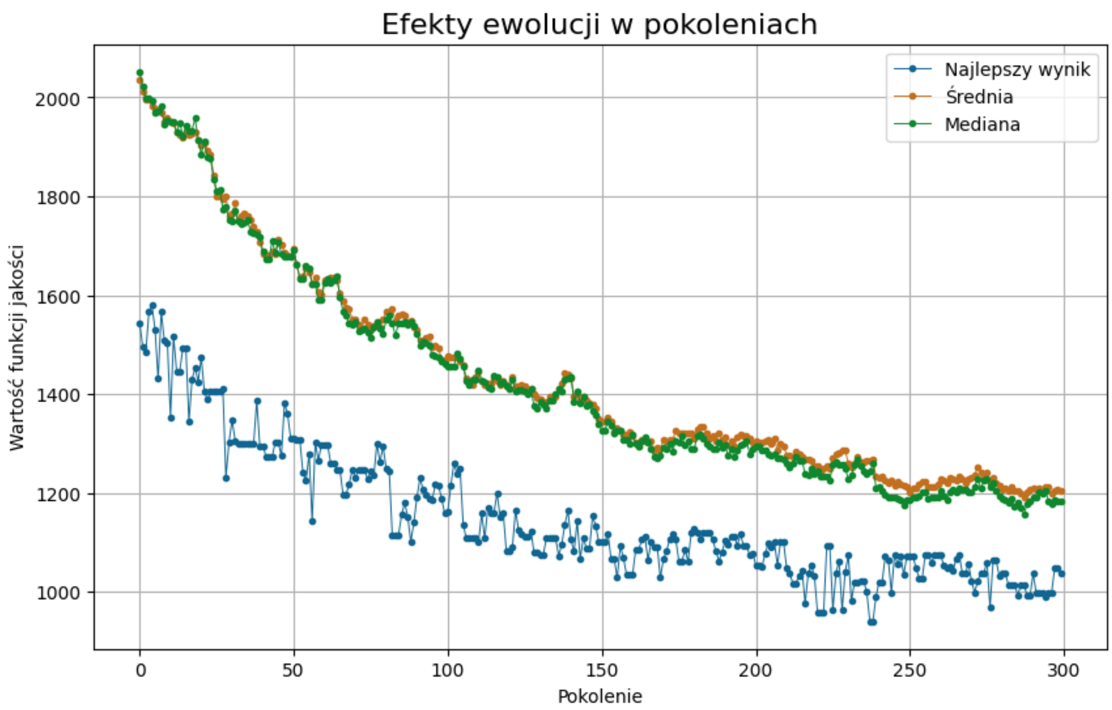



```
Najlepsza znaleziona trasa ma długość: 973.443
```

1.1.7 Zwiększenie liczebności populacji z 50 do 150

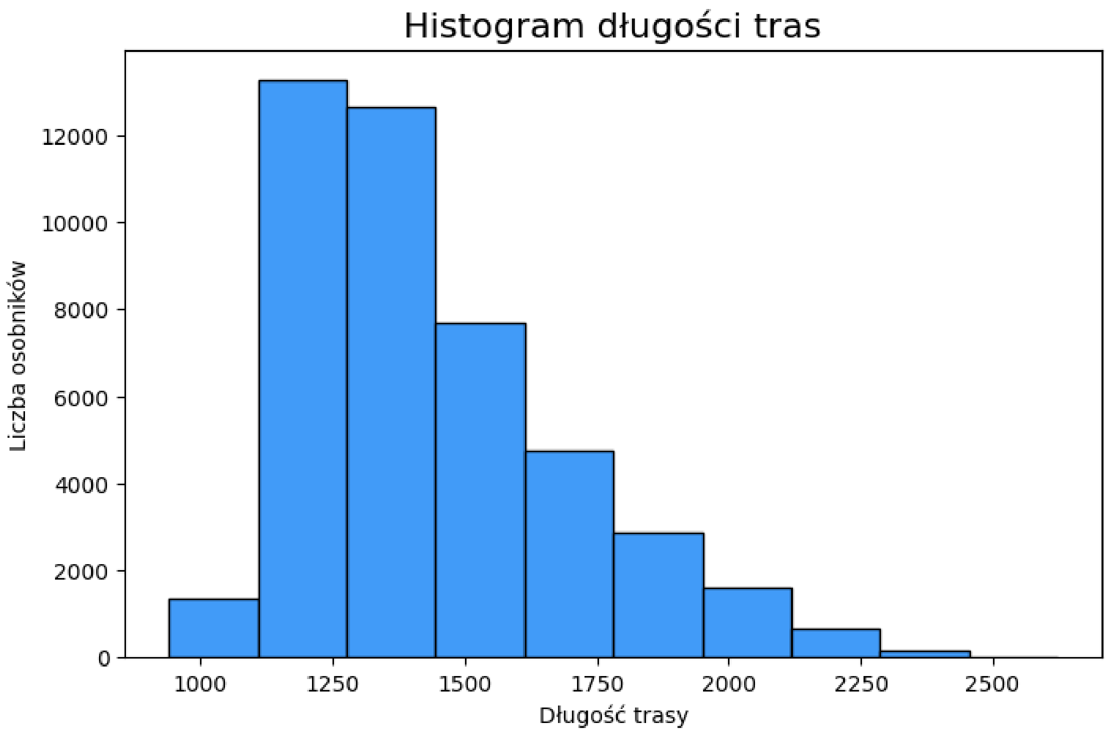
[21]:
```
best_individual, best_distance, all_fitness_scores,␣
↪best_fitness_over_generations, avg_fitness_over_generations,␣
↪median_fitness_over_generations, cities = genetic_algorithm(num_cities=20,␣
↪num_individuals=150, generations=300, mutation_rate=0.02, crossover_rate=0.
↪95)
best_route(best_individual, cities)
plot_evolution(best_fitness_over_generations, avg_fitness_over_generations,␣
↪median_fitness_over_generations, generations=300)
print(f'Najlepsza znaleziona trasa ma długość: {best_distance:.3f}')
```


Najlepsza wyznaczona trasa w generacji
Y
X
0
25
50
75
100
125
150
175
200



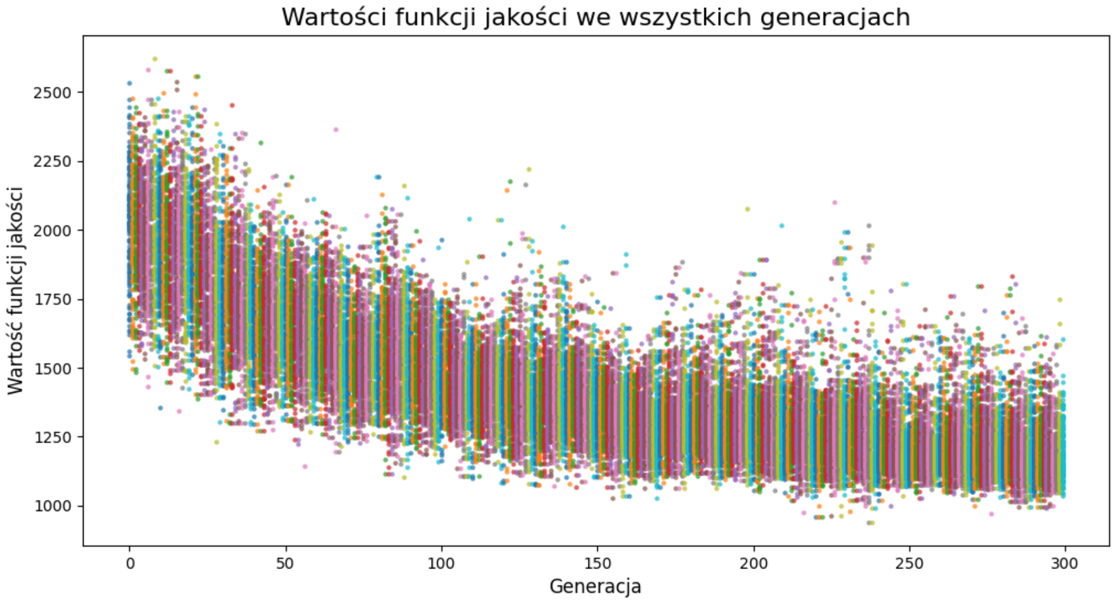
Najlepsza znaleziona trasa ma długość: 1037.173

[22]:
```
from functools import reduce
plt.figure(figsize=(8, 5))
plt.hist(list(reduce(lambda x, y: x + y, all_fitness_scores,␣
↪[])),color="#419bf8", edgecolor="black")
plt.xlabel('Długość trasy')
plt.ylabel('Liczba osobników')
plt.title('Histogram długości tras', fontsize=16)
plt.show()
```



[23]:
```
plt.figure(figsize=(12, 6))
for i, gen in enumerate(all_fitness_scores):
    plt.scatter([i]*len(gen), gen, s=5, alpha=0.6)
plt.title("Wartości funkcji jakości we wszystkich generacjach", fontsize=16)
plt.xlabel("Generacja", fontsize=12)
plt.ylabel("Wartość funkcji jakości", fontsize=12)
```

[23]:
```
Text(0, 0.5, 'Wartość funkcji jakości')
```



1.1.8 Wnioski

Algorytm został użyty w kilku różnych konfiguracjach. Możliwa była zmiana wielu parametrów. Głównie zmieniane były parametry dotyczące prawdopodobieństwa mutacji oraz krzyżowania.

Zauważalna jest optymalizacja kolejnych rozwiązań. W przypadku selekcji metodą ruletkową najlepsze wyniki uzyskano wykorzystując mutation_rate=0.2 oraz crossover_rate=0.95. Najlepszy wynik został uzyskany po zwiększeniu liczebności populacji. Możliwa byłą również zmiana metody selekcji na rankingową. Nie ma pewności czy została ona dobrze przeprowadzona, ponieważ wynik jest zaskakująco dobry. Miało to na celu sprawdzenie innych metod modyfikacji wynikami algorytmu.