algorytmgenetycznyroksanajandura

May 14, 2025

1 Algorytm Genetyczny - Problem Komiwojażera

1.0.1 Roksana Jandura

[38]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import time

%matplotlib inline
np.random.seed(42)
```

1.0.2 Funkcje pomocnicze

Funkcja generuje losowe współrzędne n_cities miast na płaszczyźnie o rozmiarze size × size. Każde miasto reprezentowane jest jako punkt (x, y). Współrzędne są losowane równomiernie w zadanym obszarze.

[3]:
```
def generate_cities(n_cities, size=300):
    return np.random.rand(n_cities, 2) * size
```

Funkcja oblicza odległość euklidesową między dwoma punktami city1 i city2.Wynik to długość odcinka łączącego dwa miasta.

[4]:
```
def calculate_distance(city1, city2):
    return np.sqrt((city1[0] - city2[0])**2 + (city1[1] - city2[1])**2)
```

Funkcja oblicza całkowitą długość trasy wyznaczonej przez permutację route. Suma odległości liczona jest po kolei między sąsiednimi miastami na trasie, a trasa jest domykana (ostatnie miasto łączy się z pierwszym). Funkcja zwraca całkowity koszt przejścia danej ścieżki.

[5]:
```
def total_distance(route, cities):
    dist = 0
    for i in range(len(route)):
        dist += calculate_distance(cities[route[i]], cities[route[(i + 1) %␣
 ↪len(route)]])
    return dist
```

Funkcja generuje początkową populację pop_size osobników. Każdy osobnik to losowa permutacja wszystkich miast (n_cities). Permutacje tworzą różne potencjalne rozwiązania problemu.

[6]:
```
def create_population(pop_size, n_cities):
    return [np.random.permutation(n_cities) for _ in range(pop_size)]
```

Funkcja ocenia jakość każdego osobnika w populacji, obliczając długość jego trasy. Zwraca wektor wartości funkcji celu (łącznych odległości) dla wszystkich osobników. Funkcja wykorzystuje total_distance dla każdej permutacji.

[7]:
```
def evaluate_population(population, cities):
    return np.array([total_distance(ind, cities) for ind in population])
```

Funkcja realizuje selekcję rankingową osobników. Osobniki są sortowane rosnąco według wartości funkcji celu (długości trasy). Najlepsze osobniki są wybierane do dalszego krzyżowania.

[8]:
```
def ranking_selection(population, scores):
    sorted_indices = np.argsort(scores)
    selected = [population[i] for i in sorted_indices[:len(population)]]
    return selected
```

Funkcja realizuje selekcję ruletkową na podstawie wartości funkcji celu. Szanse na wybór danego osobnika są proporcjonalne do odwrotności jego wyniku (lepszy osobnik = większa szansa). Losowanie odbywa się z uwzględnieniem prawdopodobieństw.

[9]:
```
def roulette_selection(population, scores):
    fitness = 1 / (scores + 1e-10)
    probabilities = fitness / np.sum(fitness)
    selected_indices = np.random.choice(len(population),
                                        size=len(population), replace=True,
                                        p=probabilities)
    selected = [population[i] for i in selected_indices]
    return selected
```

Funkcja implementuje krzyżowanie uporządkowane. Fragment trasy jednego rodzica jest kopiowany do dziecka, a reszta miast jest uzupełniana w kolejności drugiego rodzica, bez powtórzeń. Krzyżowanie zapewnia spójność permutacji.

[10]:
```
def ordered_crossover(parent1, parent2):
    size = len(parent1)
    a, b = sorted(np.random.choice(range(size), 2, replace=False))
    child = [-1] * size
    child[a:b+1] = parent1[a:b+1]
    ptr = 0
    for i in range(size):
        if child[i] == -1:
            while parent2[ptr] in child:
                ptr += 1
            child[i] = parent2[ptr]
```

```
    return np.array(child)
```

Funkcja realizuje mutację poprzez zamianę miejscami dwóch losowo wybranych miast na trasie. Mutacja wprowadza różnorodność do populacji, umożliwiając eksplorację nowych rozwiązań. Operacja jest niewielką modyfikacją trasy, zachowującą permutację.

[11]:
```
def swap_mutation(route):
    route = route.copy()  # Żeby nie zmieniać oryginału
    a, b = np.random.choice(len(route), 2, replace=False)
    route[a], route[b] = route[b], route[a]
    return route
```

Funkcja realizuje sukcesję przez całkowite zastąpienie starej populacji nową generacją. Cała populacja w kolejnym pokoleniu składa się wyłącznie z potomków.

[12]:
```
def full_succession(next_generation):
    return next_generation
```

Funkcja partial_succession łączy starą populację z nową i wybiera najlepsze osobniki do kolejnej generacji. Celem jest częściowe zastąpienie populacji, zachowując część wcześniejszych dobrych rozwiązań. Ta metoda zmniejsza ryzyko utraty wartościowych tras i zwiększa stabilność ewolucji.

[13]:
```
def partial_succession(population, next_generation, cities, pop_size):
    combined = population + next_generation
    scores_combined = evaluate_population(combined, cities)
    best_indices = np.argsort(scores_combined)[:pop_size]
    new_population = [combined[i] for i in best_indices]
    return new_population
```

Funkcja elitism_succession zapewnia, że najlepszy osobnik z poprzedniego pokolenia zawsze przechodzi do następnego. Najlepsze rozwiązanie zastępuje pierwszego osobnika w nowej populacji potomków.

[14]:
```
def elitism_succession(population, next_generation, scores):
    best_idx = np.argmin(scores)
    elite = population[best_idx]
    next_generation[0] = elite.copy()  # najlepszy zastępuje pierwszego osobnika
    return next_generation
```

1.0.3 Algorytm Genetyczny

Algorytm genetyczny jest heurystyczną metodą optymalizacji inspirowaną procesami ewolucji biologicznej. Symuluje on ewolucję populacji rozwiązań poprzez operacje selekcji, krzyżowania i mutacji, aby stopniowo poprawiać jakość rozwiązań w kolejnych pokoleniach. W każdej iteracji algorytm ocenia osobniki (rozwiązania), wybiera lepsze, generuje nowe poprzez łączenie i losowe zmiany, a następnie tworzy nową populację. Celem algorytmu jest minimalizacja funkcji celu – w tym przypadku długości trasy w problemie komiwojażera.

Parametry funkcji genetic_algorithm

- n_cities — liczba miast do odwiedzenia; definiuje rozmiar problemu komiwojażera. Im więcej miast, tym trudniejsze zadanie optymalizacji.
- pop_size — liczebność populacji; określa, ile rozwiązań (tras) istnieje w każdej generacji. Większa populacja może zwiększyć szanse na znalezienie dobrego rozwiązania, ale wydłuża czas obliczeń.
- cx_prob — prawdopodobieństwo krzyżowania; oznacza szansę, że dwaj wybrani rodzice zostaną skrzyżowani, zamiast być tylko kopiowani do następnej generacji.
- mut_prob — prawdopodobieństwo mutacji; określa, jak często potomkowie są poddawani losowej mutacji, co wpływa na różnorodność genetyczną populacji.
- n_generations — maksymalna liczba pokoleń; definiuje, jak długo będzie trwała ewolucja (ile razy populacja zostanie zaktualizowana).
- selection_method — metoda selekcji; decyduje, jak wybierani są rodzice do rozmnażania („ranking" – najlepsi osobnicy, „roulette" – losowanie z wagami).
- succession_type — typ sukcesji; określa, jak tworzona jest nowa populacja po krzyżowaniu i mutacji („full" – całkowite zastąpienie, „partial" – częściowe zastąpienie, „elitism" – zachowanie najlepszego osobnika).

1.0.4 Szczegółowy przebieg działania:

Inicjalizacja populacji: - Losowane są współrzędne miast (`generate_cities`).

- Tworzona jest początkowa populacja losowych tras (`create_population`).

Pętla główna dla kolejnych pokoleń (`for generation in range(n_generations)`):

Ocena populacji: - Obliczana jest długość trasy dla każdego osobnika (`evaluate_population`).

- Zapisywana jest najlepsza, średnia i najgorsza długość trasy do dalszej analizy.
- Aktualizowana jest najlepsza dotychczas znaleziona trasa.

Selekcja rodziców: - W zależności od parametru `selection_method`, wybierane są osobniki:

- `ranking_selection` (najlepsze wyniki),
- `roulette_selection` (prawdopodobieństwo zależne od jakości).

Krzyżowanie i mutacja: - Tworzone są nowe osobniki poprzez krzyżowanie (`ordered_crossover`) dwóch rodziców z prawdopodobieństwem `cx_prob`.

- Następnie każde dziecko może ulec mutacji (`swap_mutation`) z prawdopodobieństwem `mut_prob`.

Sukcesja (tworzenie nowej populacji): - Zależnie od parametru `succession_type`, tworzona jest nowa populacja:

- `full_succession` – całkowite zastąpienie,
- `partial_succession` – połowa starych + połowa nowych,
- `elitism_succession` – najlepszy osobnik zawsze przechodzi do kolejnego pokolenia.

Zwracane wyniki: - Funkcja zwraca listę miast, najlepszą trasę oraz historię najlepszych, średnich i najgorszych wyników w kolejnych pokoleniach.

[27]:
```
def genetic_algorithm(n_cities=30, pop_size=100, cx_prob=0.9, mut_prob=0.05,
                      n_generations=300, selection_method="ranking",
                     succession_type="full"):
    cities = generate_cities(n_cities)
    population = create_population(pop_size, n_cities)

    best_scores = []
    avg_scores = []
    worst_scores = []
    best_score = np.inf
    best_route = None
    best_routes_per_generation = []
    best_generation = None

    for generation in range(n_generations):
        scores = evaluate_population(population, cities)

        if np.min(scores) < best_score:
            best_score = np.min(scores)
            best_route = population[np.argmin(scores)]
            best_generation = generation

        best_scores.append(np.min(scores))
        avg_scores.append(np.mean(scores))
        worst_scores.append(np.max(scores))
        best_routes_per_generation.append(best_route.copy())



        # Selekcja
        if selection_method == "ranking":
            selected = ranking_selection(population, scores)
        elif selection_method == "roulette":
            selected = roulette_selection(population, scores)
        else:
            raise ValueError("Unknown selection method: choose 'ranking' or␣
↪'roulette'.")

        # Krzyżowanie + Mutacja
        children = []
        for i in range(0, len(selected), 2):
            parent1 = selected[i]
            parent2 = selected[(i+1) % len(selected)]
            if np.random.rand() < cx_prob:
                child1 = ordered_crossover(parent1, parent2)
                child2 = ordered_crossover(parent2, parent1)
            else:
```

```
                child1, child2 = parent1.copy(), parent2.copy()
            children.append(child1)
            children.append(child2)

        next_generation = []
        for child in children:
            if np.random.rand() < mut_prob:
                child = swap_mutation(child)
            next_generation.append(child)


        # Sukcesja
        if succession_type == "full":
            population = full_succession(next_generation)
        elif succession_type == "partial":
            population = partial_succession(population, next_generation,␣
↪cities, pop_size)
        elif succession_type == "elitism":
            population = elitism_succession(population, next_generation, scores)
        else:
            raise ValueError("Unknown succession_type: choose 'full', 'partial'␣
↪or 'elitism'.")

    return cities, best_route, best_scores, avg_scores, worst_scores,␣
↪best_routes_per_generation, best_generation, best_score
```

1.0.5 Wizualizacje i wnioski

plot_route(cities, route, title="Najlepsza trasa") - Funkcja wizualizuje najlepszą znalezioną trasę spośród wszystkich pokoleń, łącząc miasta liniami na wykresie. Miasta oznaczane są czerwonymi punktami, a trasa – niebieskimi odcinkami. Wykres przedstawia rozwiązanie problemu komiwojażera w układzie współrzędnych, prezentując globalnie najlepsze osiągnięte rozwiązanie

plot_scores(best_scores, avg_scores, worst_scores) - funkcja przedstawia wykres zmian jakości rozwiązań w kolejnych pokoleniach. Rysuje trzy krzywe: najlepszego, średniego i najgorszego osobnika w każdej generacji. Pozwala analizować przebieg procesu ewolucji i skuteczność algorytmu.

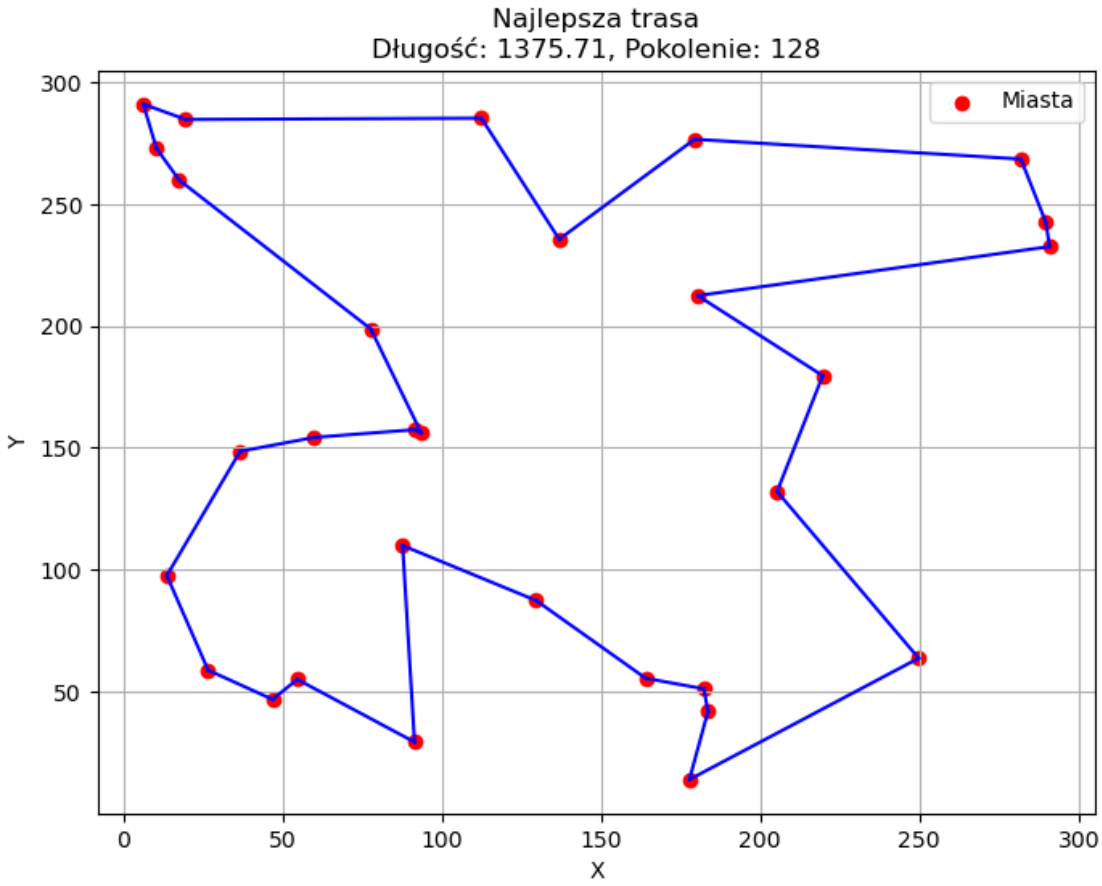
[28]:
```
def plot_route(cities, route, generation=None, distance=None):
    plt.figure(figsize=(8, 6))
    plt.scatter(cities[:, 0], cities[:, 1], c='red', label="Miasta")
    for i in range(len(route)):
        city1 = cities[route[i]]
        city2 = cities[route[(i + 1) % len(route)]]
        plt.plot([city1[0], city2[0]], [city1[1], city2[1]], 'b-')

    title = "Najlepsza trasa"
    if distance is not None and generation is not None:
```

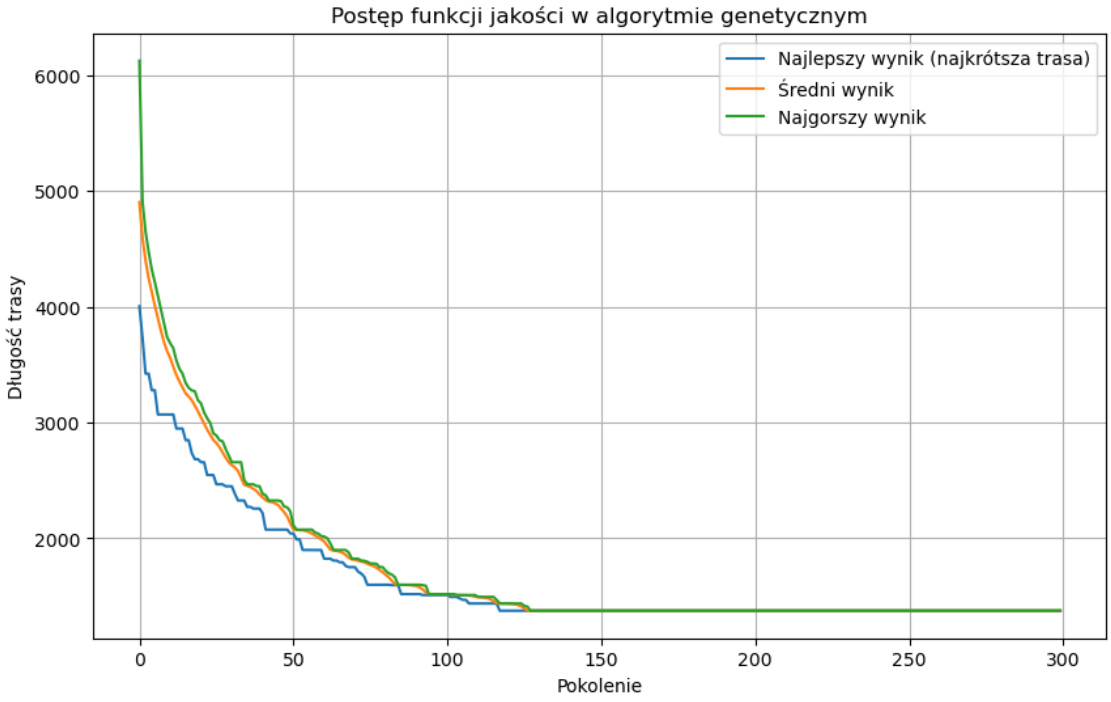
```
        title += f"\nDługość: {distance:.2f}, Pokolenie: {generation}"

    plt.title(title)
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.legend()
    plt.grid(True)
    plt.show()



def plot_scores(best_scores, avg_scores, worst_scores):
    plt.figure(figsize=(10, 6))
    plt.plot(best_scores, label='Najlepszy wynik (najkrótsza trasa)')
    plt.plot(avg_scores, label='Średni wynik')
    plt.plot(worst_scores, label='Najgorszy wynik')
    plt.xlabel('Pokolenie')
    plt.ylabel('Długość trasy')
    plt.title('Postęp funkcji jakości w algorytmie genetycznym')
    plt.legend()
    plt.grid(True)
    plt.show()
```

Wstępne analizy

Na początku eksperymentu przyjęto różne zestawy parametrów algorytmu genetycznego, takie jak liczba miast, rozmiar populacji, prawdopodobieństwo krzyżowania i mutacji oraz liczba generacji. Celem było przeprowadzenie wstępnej analizy, aby lepiej zrozumieć, jak zmiana poszczególnych parametrów wpływa na zachowanie algorytmu, jakość rozwiązań oraz tempo konwergencji. Pozwoliło to na obserwację ogólnych trendów działania algorytmu i identyfikację ustawień sprzyjających szybszemu lub bardziej stabilnemu znajdowaniu krótszych tras.

[31]:
```
np.random.seed(42) # dla lepszej wygody opisu wizualizacji
cities, best_route, best_scores, avg_scores, worst_scores,␣
 ↪best_routes_per_generation, best_generation, best_score = genetic_algorithm(
    n_cities=30,
    pop_size=300,
    cx_prob=0.9,
    mut_prob=0.2,
    n_generations=300,
    selection_method="ranking",
    succession_type="partial"
)

plot_route(cities, best_route, generation=best_generation, distance=best_score)
plot_scores(best_scores, avg_scores, worst_scores)
```

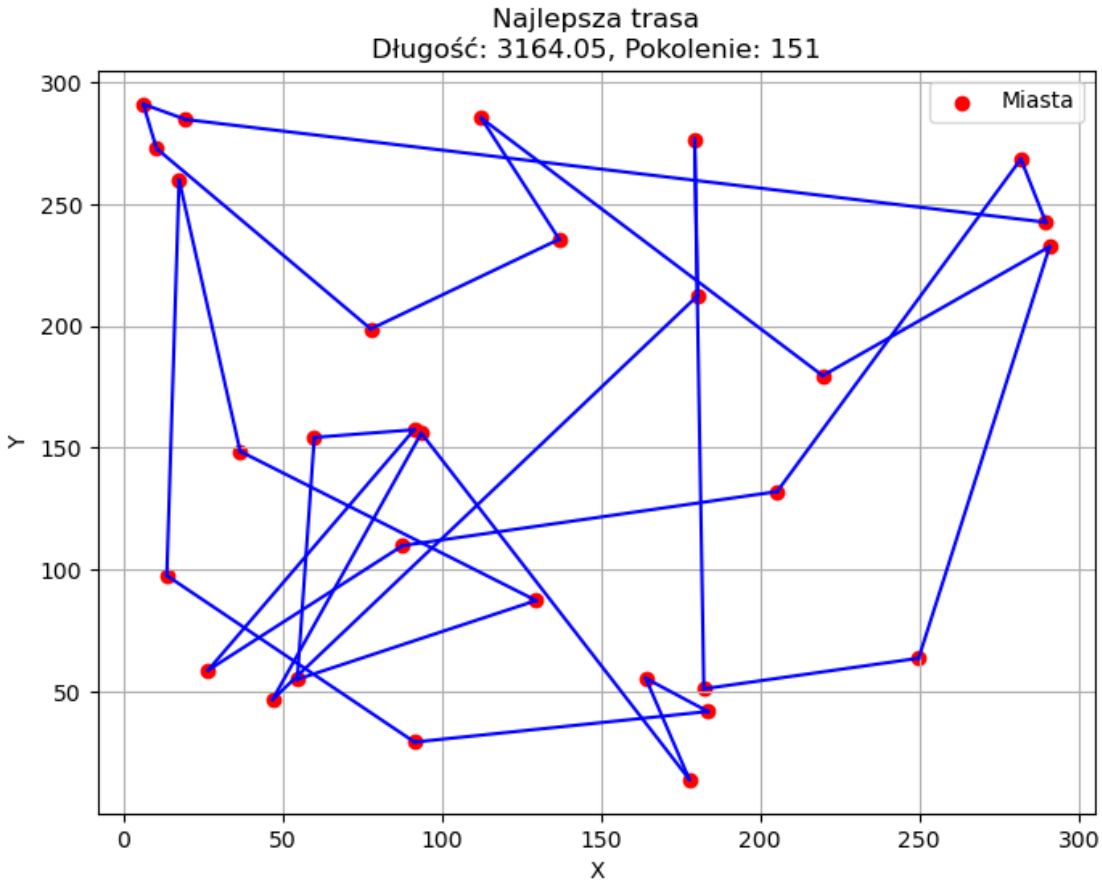

Najlepsza trasa
Długość: 1375.71, Pokolenie: 128
Miasta
300
250
200
150
100
50
Y
0
50
100
150
200
250
300
X

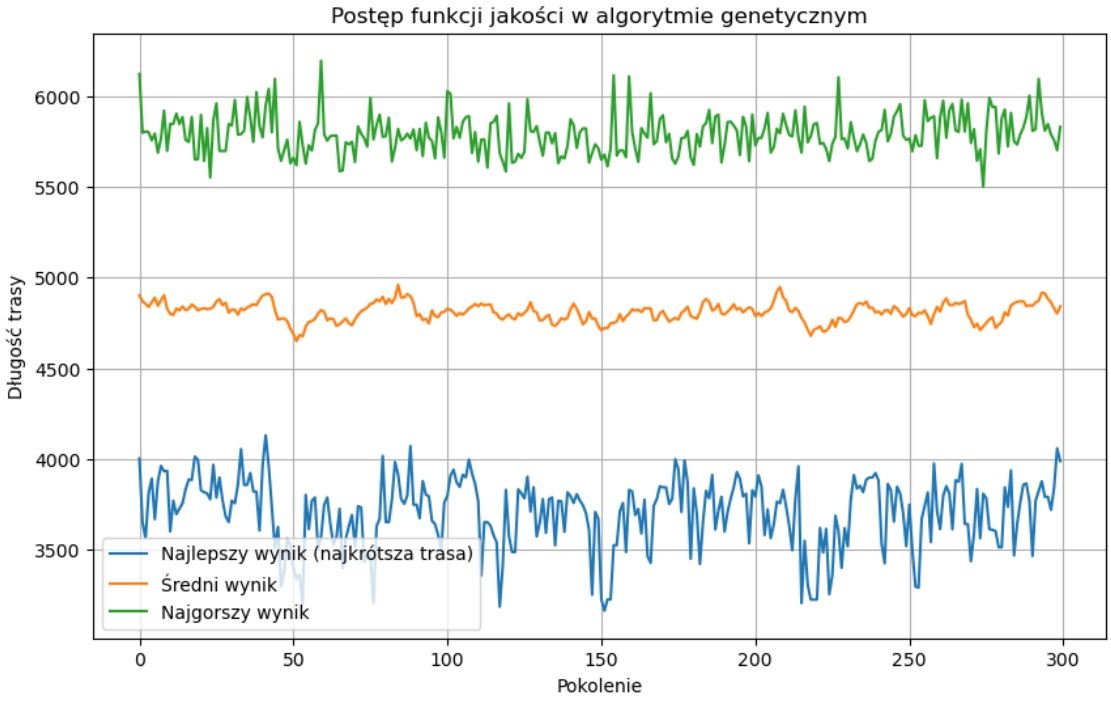


Najlepsza trasa została odnaleziona w 128. pokoleniu, osiągając długość 1375.71. Ścieżka pomiędzy miastami jest dobrze zoptymalizowana – odległości między kolejnymi punktami są relatywnie krótkie, a trasa unika zbędnych powrotów lub dużych "przeskoków". Na początku algorytm bardzo szybko poprawiał jakość tras (gwałtowny spadek funkcji celu). Około 100. pokolenia tempo poprawy zwolniło, a krzywe dla najlepszego, średniego i najgorszego wyniku zaczęły się stabilizować. Widzimy, że średni wynik populacji stale się poprawiał, co świadczy o efektywnej selekcji rankingowej i stabilnym działaniu algorytmu przy sukcesji częściowej.

Użycie selekcji rankingowej i częściowej sukcesji przy dużej populacji (300) pozwoliło na skuteczne i szybkie znalezienie bardzo dobrej trasy. Mutacja na poziomie 20% pomogła uniknąć utknięcia w lokalnych minimach, co widać po ciągłej poprawie średniego wyniku.

[45]:
```
np.random.seed(42)
cities, best_route, best_scores, avg_scores, worst_scores,␣
 ↪best_routes_per_generation, best_generation, best_score = genetic_algorithm(
    n_cities=30,
    pop_size=300,
    cx_prob=0.9,
    mut_prob=0.2,
    n_generations=300,
    selection_method="roulette",     # zmiana na ruletkę
    succession_type="full"           # pełne zastępowanie
)
plot_route(cities, best_route, generation=best_generation, distance=best_score)
plot_scores(best_scores, avg_scores, worst_scores)
```

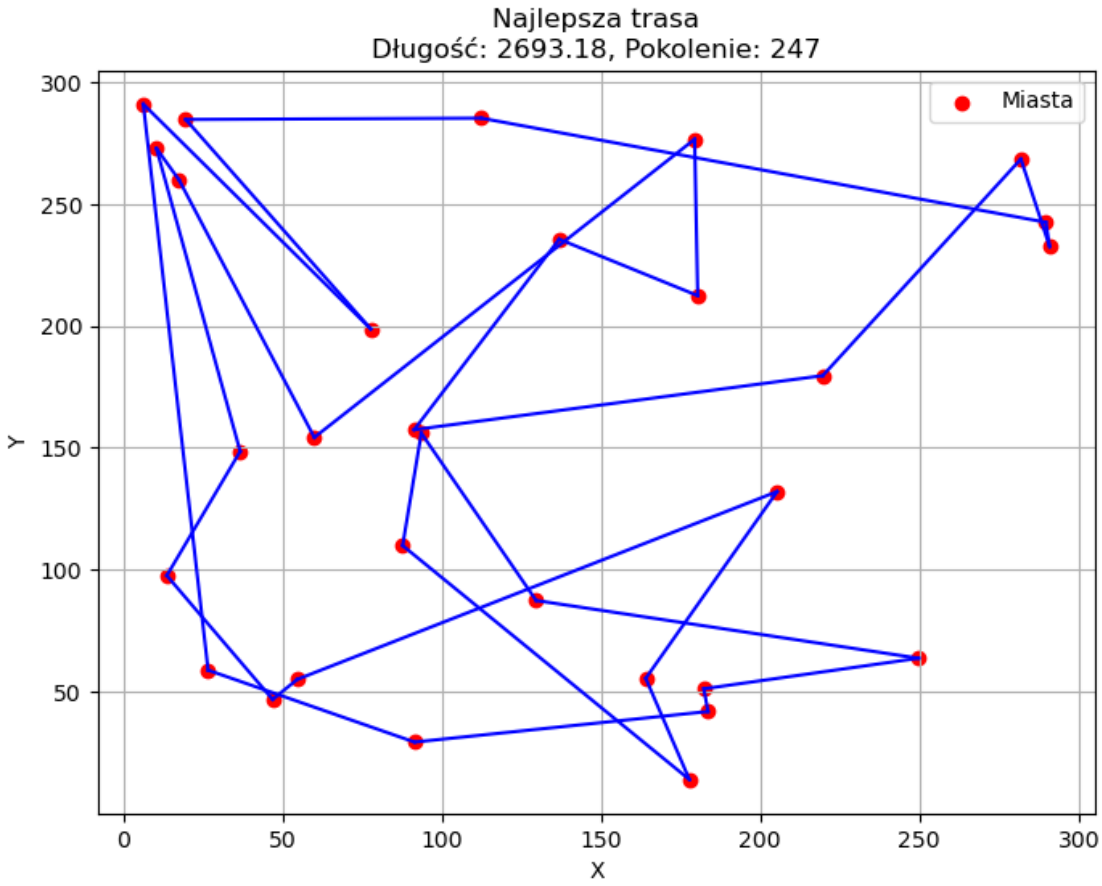

Najlepsza trasa
Długość: 3164.05, Pokolenie: 151
Miasta
300
250
200
150
100
50
Y
0
50
100
150
200
250
300
X

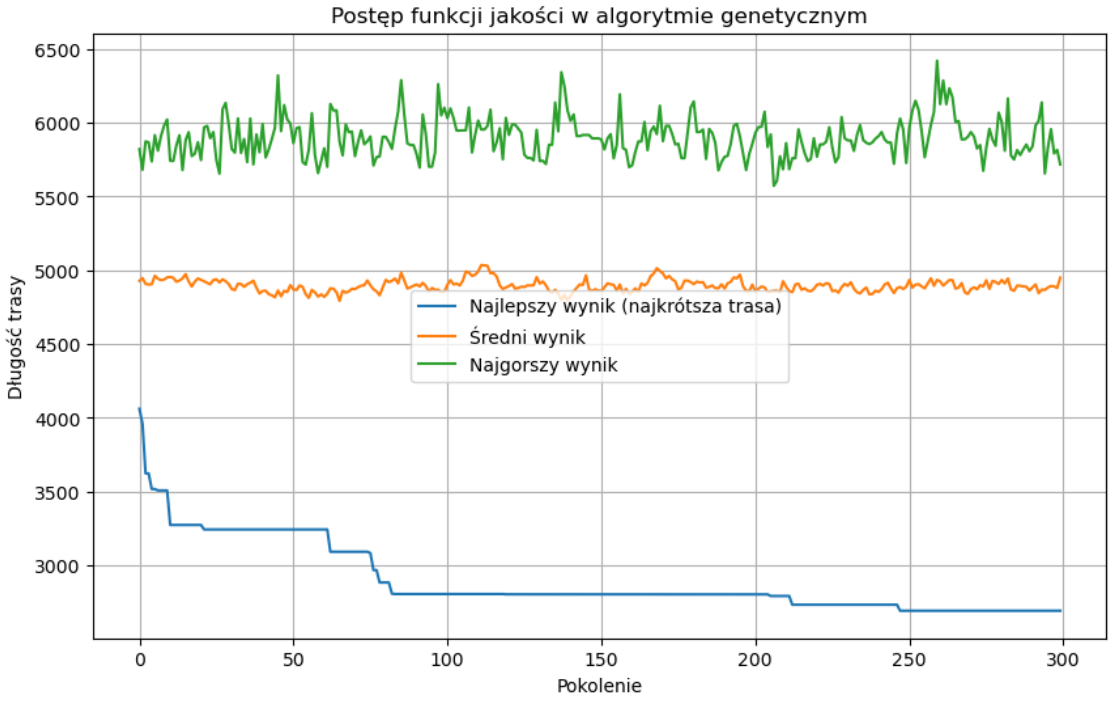


Najlepsza znaleziona trasa ma długość około 3164 jednostek i została odnaleziona w 151. pokoleniu. Ścieżka jest wyraźnie nieoptymalna – widać wiele przeskoków między miastami, a trasa przecina się wielokrotnie, co jest charakterystyczne dla słabej jakości rozwiązania.Krzywa najlepszego wyniku (niebieska) waha się znacząco przez całą ewolucję, bez wyraźnej poprawy.

Średnie i najgorsze wyniki również pozostają na wysokim i dość stałym poziomie, co świadczy o braku skutecznej poprawy populacji. Pełna sukcesja (full) oraz ruletkowa selekcja prowadzą do dużej zmienności i braku stabilności – algorytm losowo błądzi, nie budując stopniowo lepszych tras.

[34]:
```
np.random.seed(42)
cities, best_route, best_scores, avg_scores, worst_scores,␣
 ↪best_routes_per_generation, best_generation, best_score = genetic_algorithm(
    n_cities=30,
    pop_size=100,                    # mniejsza populacja
    cx_prob=0.9,
    mut_prob=0.2,
    n_generations=300,
    selection_method="ranking",
    succession_type="elitism"         # elitarna sukcesja
)
plot_route(cities, best_route, generation=best_generation, distance=best_score)
plot_scores(best_scores, avg_scores, worst_scores)
```


Najlepsza trasa
Długość: 2693.18, Pokolenie: 247
Miasta
X
Y
0
50
100
150
200
250
300



W kolejnych pokoleniach obserwujemy wyraźną poprawę najlepszego rozwiązania, choć tempo tej poprawy spowalnia w drugiej połowie ewolucji. Funkcja najlepszego wyniku (niebieska linia) systematycznie maleje, osiągając wartość około 2693 w 247. pokoleniu. Średnia i najgorsza jakość populacji (pomarańczowa i zielona linia) pozostają natomiast na względnie stałym poziomie, co wskazuje na brak istotnej poprawy całej populacji. Analiza najlepszej znalezionej trasy ujawnia pewne nielogiczne przeskoki, sugerujące, że przy małej liczebności populacji i umiarkowanej mutacji algorytm zbliżył się do minimum lokalnego, ale nie osiągnął rozwiązania optymalnego.

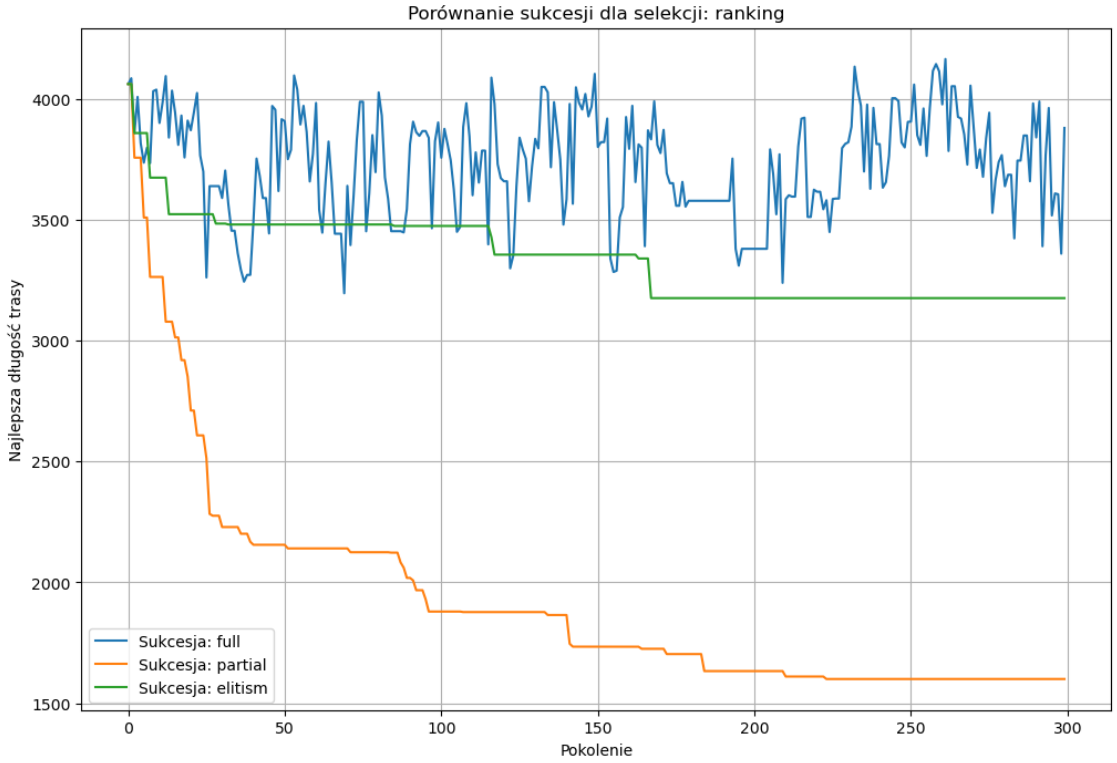
Porównanie różnych typów sukcesji:

W celu porównania skuteczności różnych metod sukcesji w algorytmie genetycznym, przygotowano eksperymenty dla obu typów selekcji: rankingowej i ruletkowej. Dla każdej selekcji analizowano trzy rodzaje sukcesji: pełną, częściową oraz elitarną. Pozwoliło to ocenić, jak wybór strategii sukcesji wpływa na tempo i jakość zbieżności algorytmu. W każdym eksperymencie użyto tych samych parametrów algorytmu (m.in. liczba miast, wielkość populacji, liczba pokoleń, metoda selekcji), aby uzyskane wyniki były bezpośrednio porównywalne. Na wykresie przedstawiono przebieg najlepszego wyniku w kolejnych pokoleniach dla każdej strategii sukcesji.

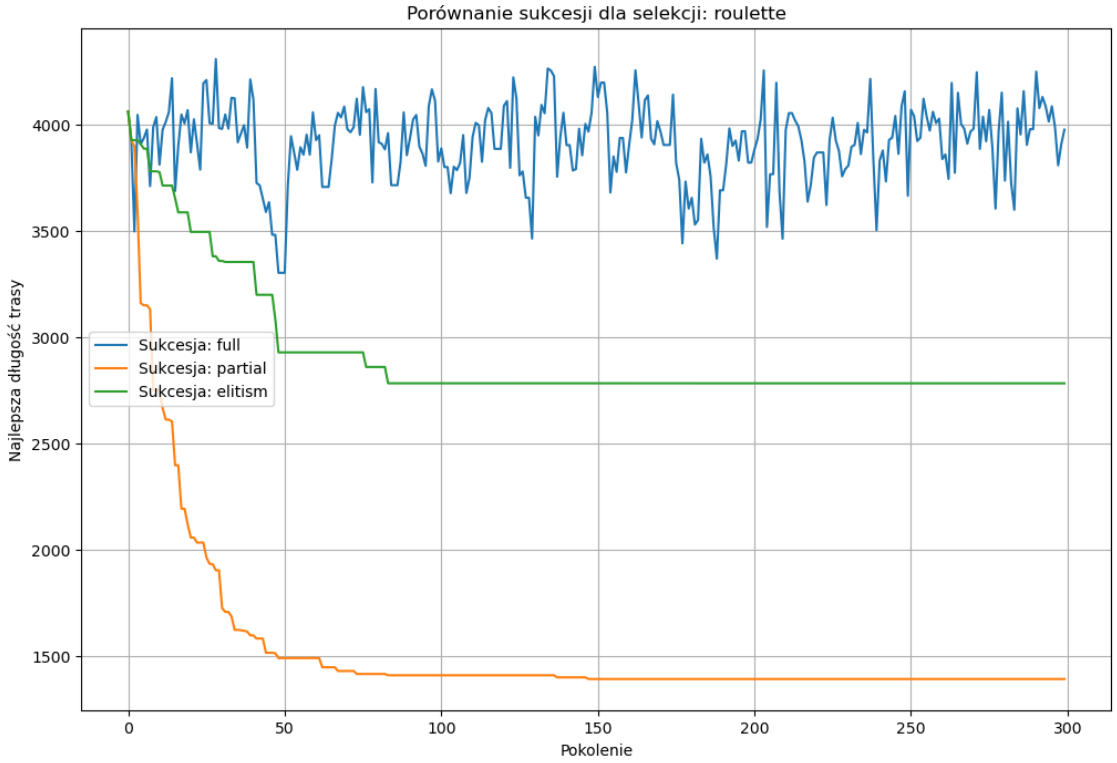
[37]:
```
def compare_succession_for_each_selection(selection_methods, succession_types,␣
↪common_params, seed=42):
    for method in selection_methods:
        results = []

        for succession in succession_types:
            np.random.seed(seed)  # żeby zawsze warunki były takie same
```

```
            print(f"Testuję selekcję: {method}, sukcesja: {succession}")
            cities, best_route, best_scores, avg_scores, worst_scores,␣
 ↪best_routes_per_generation, best_generation, best_score = genetic_algorithm(
                **common_params,
                selection_method=method,
                succession_type=succession
            )
            results.append((succession, best_scores))

        plt.figure(figsize=(12, 8))

        for succession, best_scores in results:
            plt.plot(best_scores, label=f"Sukcesja: {succession}")

        plt.xlabel('Pokolenie')
        plt.ylabel('Najlepsza długość trasy')
        plt.title(f'Porównanie sukcesji dla selekcji: {method}')
        plt.legend()
        plt.grid(True)
        plt.show()

# Parametry wspólne
common_params = {
    "n_cities": 30,
    "pop_size": 100,
    "cx_prob": 0.9,
    "mut_prob": 0.05,
    "n_generations": 300
}
selection_methods = ["ranking", "roulette"]
succession_types = ["full", "partial", "elitism"]

compare_succession_for_each_selection(selection_methods, succession_types,␣
 ↪common_params)
```

```
Testuję selekcję: ranking, sukcesja: full
Testuję selekcję: ranking, sukcesja: partial
Testuję selekcję: ranking, sukcesja: elitism
```




```
Testuję selekcję: roulette, sukcesja: full
Testuję selekcję: roulette, sukcesja: partial
Testuję selekcję: roulette, sukcesja: elitism
```



Wnioski – wykres dla **selekcji rankingowej**

W przypadku selekcji rankingowej:

- Sukcesja częściowa zapewniła najlepsze wyniki – algorytm stopniowo i stabilnie poprawiał rozwiązania, osiągając najkrótsze trasy.
- Sukcesja elitarna pozwoliła utrzymać dobre rozwiązanie przez wiele pokoleń, jednak progres był wolniejszy niż przy częściowej sukcesji.
- Sukcesja pełna charakteryzowała się dużą zmiennością wyników i brakiem stabilnej poprawy, co wynika z braku zachowania dobrych osobników.

Wnioski – wykres dla **selekcji ruletkowej**

W przypadku selekcji ruletkowej:

- Sukcesja częściowa ponownie okazała się najlepsza, prowadząc do szybkiego i stabilnego spadku długości tras.
- Sukcesja elitarna zapewniła lepsze wyniki niż pełne zastępowanie, ale osiągnięta jakość była nieco niższa niż przy selekcji rankingowej.
- Sukcesja pełna wykazywała największe wahania wyników, bez wyraźnego trendu poprawy.

Wnioski:

- Sukcesja częściowa w obu metodach selekcji zdecydowanie wypada najlepiej – zapewnia równowagę między zachowaniem dobrych rozwiązań a możliwością eksploracji nowych.
- Sukcesja elitarna jest bezpieczna, ale może powodować przedwczesne zatrzymanie algorytmu w lokalnym minimum.
- Sukcesja pełna jest najmniej stabilna i wprowadza dużo losowości, co znacząco utrudnia osiągnięcie dobrych wyników.

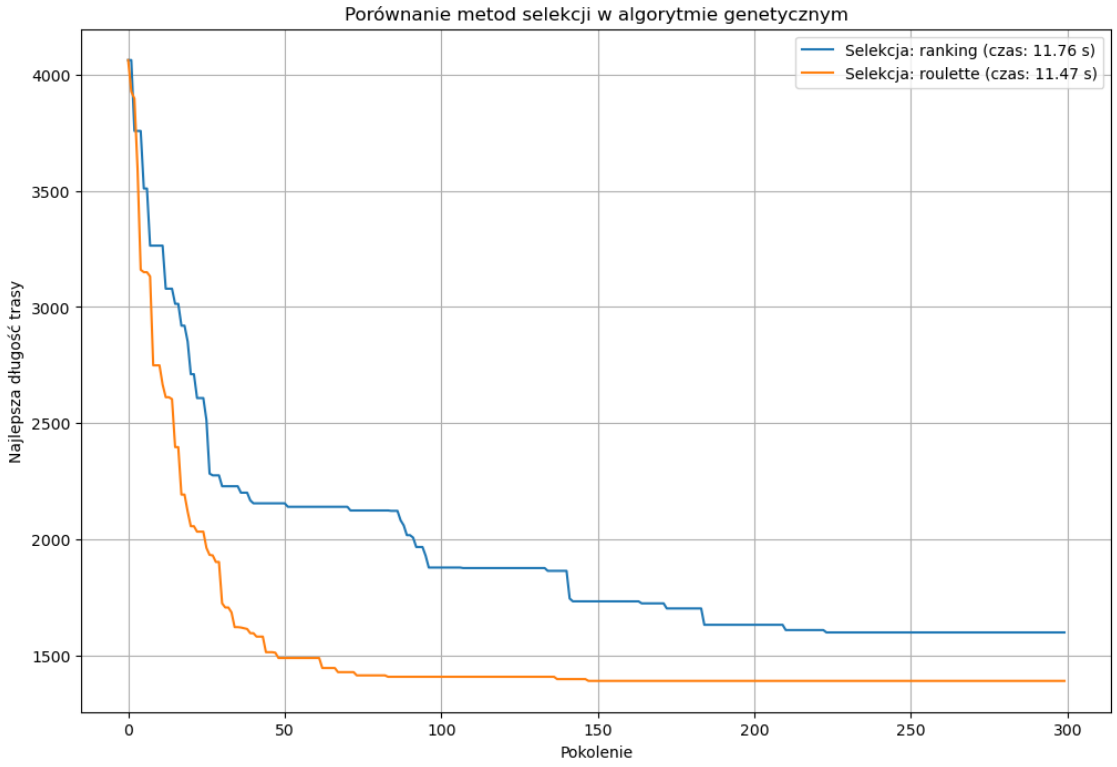
Porównanie metod selekcji wraz z czasem wykonania

W tej części pracy przeprowadzono porównanie metod selekcji w algorytmie genetycznym – rankingowej i ruletkowej – pod kątem dwóch kryteriów: jakości najlepszego znalezionego rozwiązania oraz czasu działania algorytmu. W obu przypadkach wykorzystano te same parametry eksperymentu: 30 miast, populację 100 osobników, prawdopodobieństwo krzyżowania 0.9, mutacji 0.05, 300 pokoleń oraz sukcesję częściową (partial). Funkcja compare_selection_methods automatycznie mierzy czas działania każdej metody oraz wizualizuje postęp najlepszych rozwiązań w kolejnych pokoleniach.

[46]:
```
def compare_selection_methods(genetic_algorithm_func, common_params,␣
↪selection_methods):
    results = []

    for selection in selection_methods:
        print(f"Testuję selekcję: {selection}")
        np.random.seed(42)
        start_time = time.time()

        cities, best_route, best_scores, avg_scores, worst_scores,␣
↪best_routes_per_generation, best_generation, best_score =␣
↪genetic_algorithm_func(
            **common_params,
            selection_method=selection
        )

        end_time = time.time()
        elapsed_time = end_time - start_time

        results.append((selection, best_scores, elapsed_time))

    plt.figure(figsize=(12, 8))

    for selection, best_scores, elapsed_time in results:
        plt.plot(best_scores, label=f"Selekcja: {selection} (czas:␣
↪{elapsed_time:.2f} s)")

    plt.xlabel('Pokolenie')
    plt.ylabel('Najlepsza długość trasy')
    plt.title('Porównanie metod selekcji w algorytmie genetycznym')
```

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

common_params = {
    "n_cities": 30,
    "pop_size": 100,
    "cx_prob": 0.9,
    "mut_prob": 0.05,
    "n_generations": 300,
    "succession_type": "partial"
}

compare_selection_methods(genetic_algorithm, common_params, selection_methods)
```

```
Testuję selekcję: ranking
Testuję selekcję: roulette
```



Metoda ruletkowa pozwala szybciej osiągnąć krótsze długości tras niż selekcja rankingowa.

Dla ruletki najlepsze rozwiązania pojawiają się już na wcześniejszych etapach ewolucji (około 50–100 pokolenia).

Ranking poprawia jakość rozwiązań wolniej i ostatecznie osiąga gorszą długość trasy.

Czas działania algorytmu jest podobny dla obu metod (różnica rzędu 0.3 s), z lekką przewagą ruletki.

Analiza wpływu wybranych parametrów algorytmu genetycznego

W celu przeanalizowania wpływu poszczególnych parametrów algorytmu genetycznego na jakość uzyskiwanych rozwiązań, przygotowano funkcję testującą zmienność wybranego parametru przy pozostałych wartościach utrzymanych na stałym poziomie. Funkcja test_parameter umożliwia obserwację, jak zmiana jednego parametru wpływa na przebieg ewolucji populacji i efektywność znajdowania najlepszej trasy.

[51]:
```
def test_parameter(param_name, values, fixed_params):
    results = []
    for value in values:
        params = fixed_params.copy()
        params[param_name] = value
        print(f"Testuję {param_name} = {value}")
        np.random.seed(42)
        cities, best_route, best_scores, avg_scores, worst_scores,␣
 ↪best_routes_per_generation, best_generation, best_score  =␣
 ↪genetic_algorithm(**params)
        results.append((value, best_scores))

    plt.figure(figsize=(12, 8))
    for value, best_scores in results:
        plt.plot(best_scores, label=f"{param_name} = {value}")
    plt.xlabel('Pokolenie')
    plt.ylabel('Najlepsza długość trasy')
    plt.title(f'Porównanie wpływu {param_name}')
    plt.legend()
    plt.grid(True)
    plt.show()
```

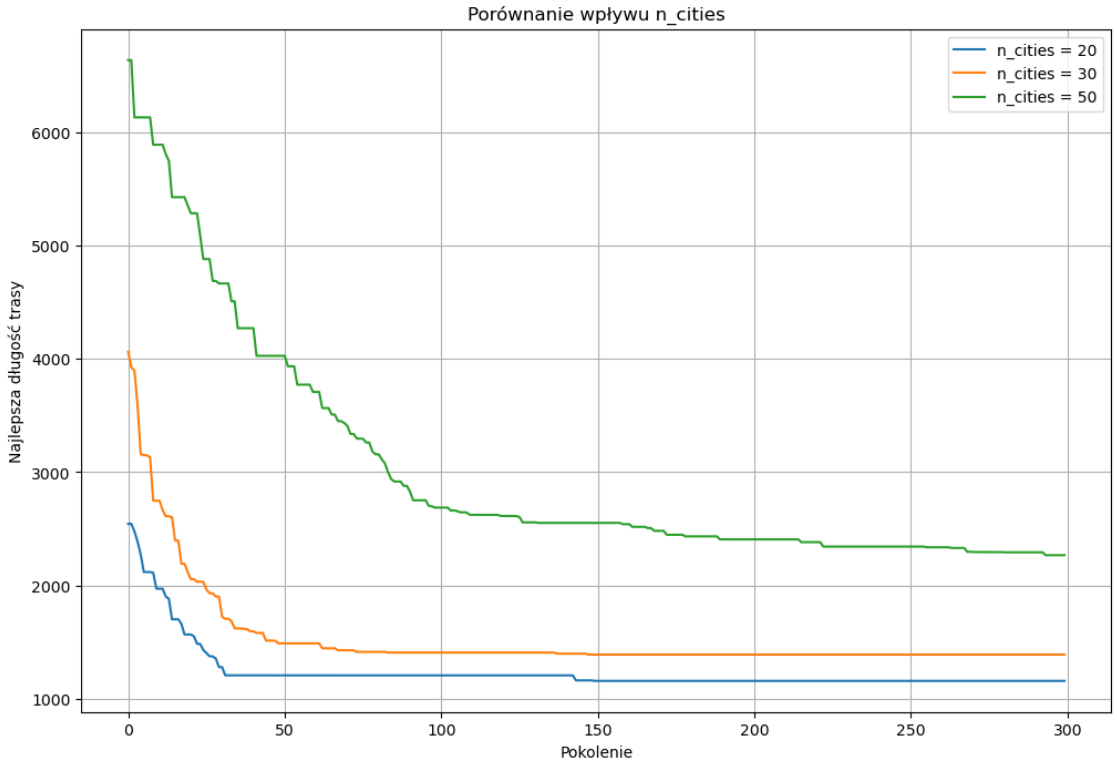
Wpływ liczby miast na jakość rozwiązania w algorytmie genetycznym

W tej części analizie poddano wpływ liczby miast na działanie algorytmu genetycznego w problemie komiwojażera. Przy stałych pozostałych parametrach zmieniano wartość n_cities, aby zaobserwować, jak zwiększenie złożoności problemu wpływa jakość osiąganych wyników. Testy przeprowadzono dla 20, 30 i 50 miast.

[52]:
```
base_params = {
    "pop_size": 100,
    "n_cities": 50,
    "cx_prob": 0.9,
    "mut_prob": 0.05,
    "n_generations": 300,
    "selection_method": "roulette",
    "succession_type": "partial"
}
```

```
test_parameter('n_cities', [20, 30, 50], base_params)
```

```
Testuję n_cities = 20
Testuję n_cities = 30
Testuję n_cities = 50
```



Wraz ze wzrostem liczby miast (n_cities) rośnie trudność problemu, co skutkuje wyższą wartością funkcji celu (dłuższą trasą).

Dla mniejszych instancji (20 miast) algorytm szybko osiąga stabilne i krótkie trasy. Dla 30 miast również obserwujemy dobrą konwergencję, ale przy nieco dłuższych trasach.

W przypadku 50 miast poprawa jakości trasy jest dużo wolniejsza, a ostateczna długość trasy pozostaje znacznie wyższa, mimo wielu pokoleń.

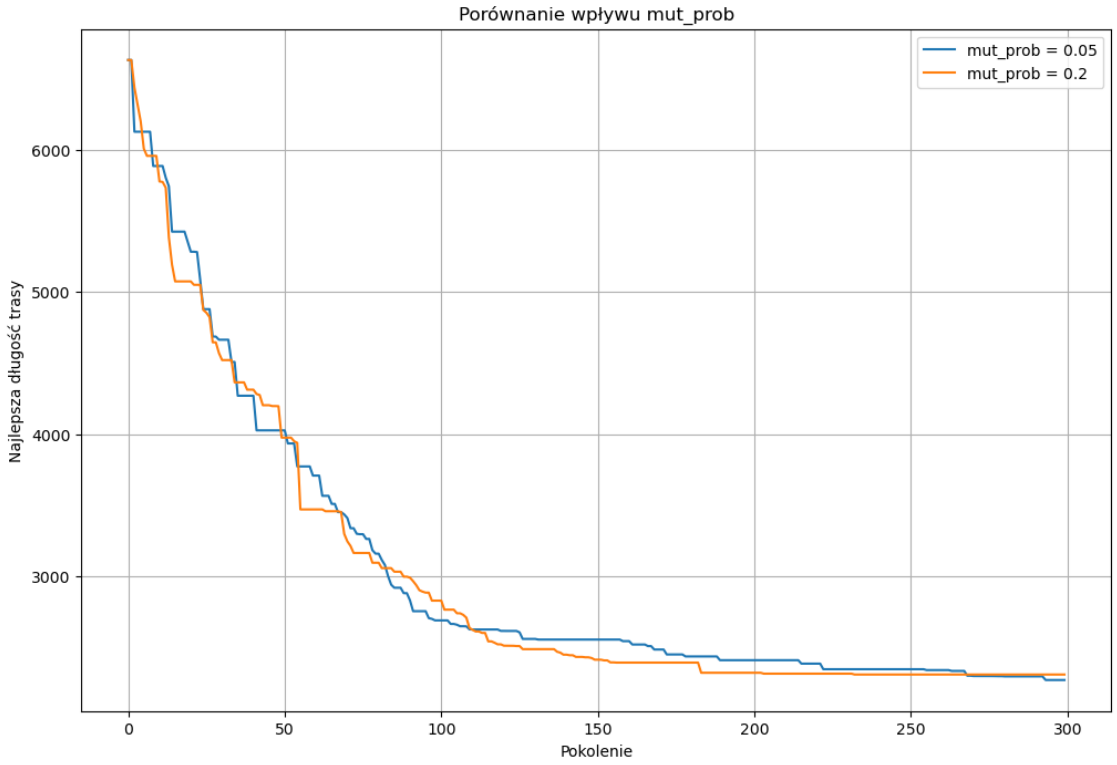
Wyniki potwierdzają, że wraz ze wzrostem skali problemu algorytm genetyczny potrzebuje więcej czasu, większej różnorodności lub intensywniejszej eksploracji, by osiągnąć rozwiązania bliskie optymalnym.

Wpływ prawdopodobieństwa mutacji na skuteczność algorytmu genetycznego

W tej części przeanalizowano wpływ zmiany wartości prawdopodobieństwa mutacji (mut_prob) na przebieg i skuteczność działania algorytmu genetycznego. Mutacja jest kluczowym mechanizmem zapewniającym różnorodność populacji, dlatego zmiana jej intensywności może znacząco wpływać na tempo oraz jakość poszukiwania optymalnego rozwiązania.

[54]:
```
test_parameter('mut_prob', [0.05, 0.2], base_params)
```

```
Testuję mut_prob = 0.05
Testuję mut_prob = 0.2
```



Wyższe prawdopodobieństwo mutacji (mut_prob = 0.2) umożliwiło osiągnięcie nieznacznie lepszych wyników (krótszych tras) w porównaniu do niższego (mut_prob = 0.05).

Przy większej mutacji obserwujemy płynniejszy spadek wartości funkcji celu w kolejnych pokoleniach, co sugeruje lepszą eksplorację przestrzeni rozwiązań.

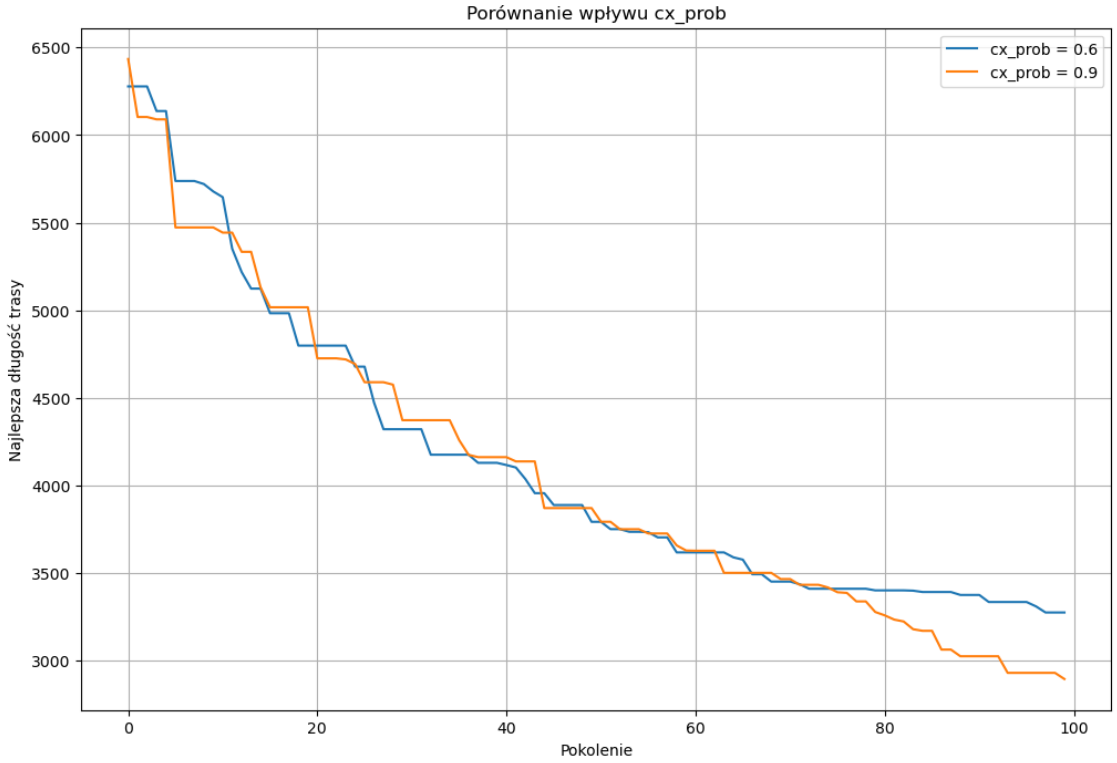
Niskie prawdopodobieństwo mutacji może powodować zbyt szybkie „utknięcie" w minimum lokalnym, bez dalszej istotnej poprawy wyniku.

Wpływ prawdopodobieństwa krzyżowania na skuteczność algorytmu genetycznego

W tej części analizie poddano wpływ wartości prawdopodobieństwa krzyżowania (cx_prob) na efektywność algorytmu genetycznego. Krzyżowanie jest podstawowym operatorem generowania nowych rozwiązań poprzez wymianę materiału genetycznego pomiędzy rodzicami, dlatego jego intensywność może decydować o szybkości i jakości poszukiwania optimum.

[22]:
```
test_parameter('cx_prob', [0.6, 0.9], base_params)
```

```
Testuję cx_prob = 0.6
Testuję cx_prob = 0.9
```



Wyższe prawdopodobieństwo krzyżowania (cx_prob = 0.9) pozwoliło na osiągnięcie krótszych tras w porównaniu do niższego (cx_prob = 0.6).

Przy wysokim cx_prob obserwujemy szybsze i stabilniejsze zmniejszanie długości trasy w kolejnych pokoleniach.

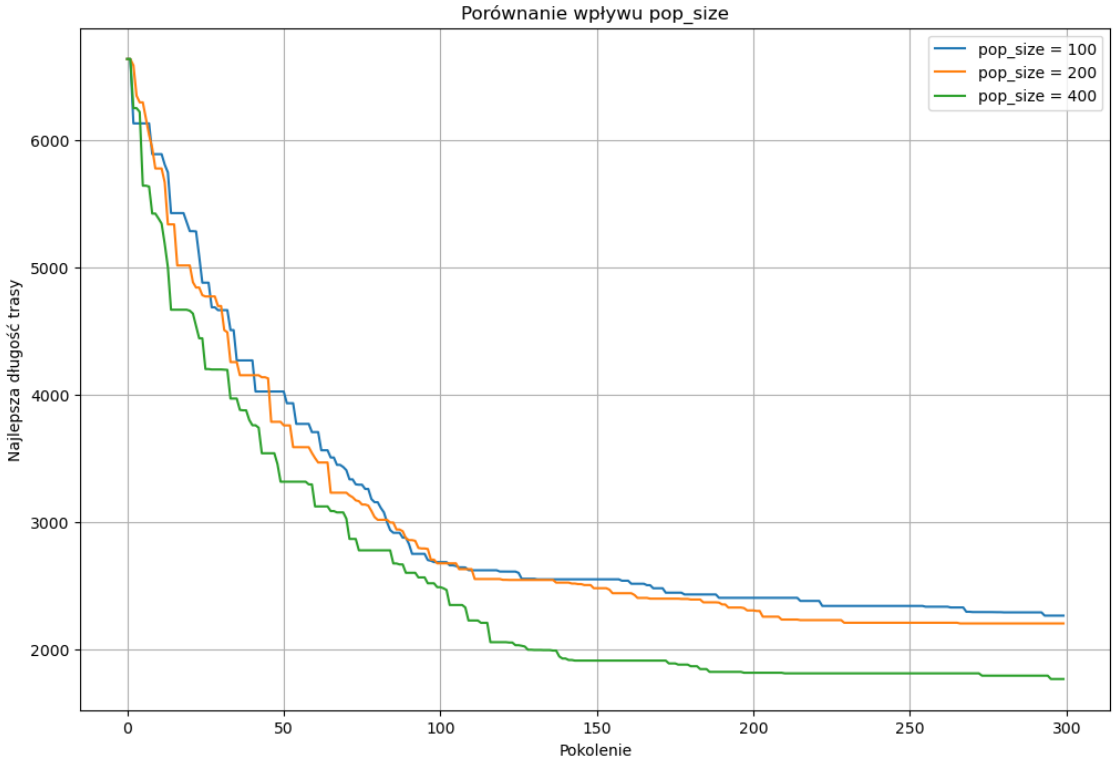
Niskie prawdopodobieństwo krzyżowania może ograniczać eksplorację przestrzeni rozwiązań, przez co algorytm wolniej poprawia jakość populacji.

Wpływ liczebności populacji na skuteczność algorytmu genetycznego

W tej części przeanalizowano wpływ parametru pop_size, czyli liczby osobników w populacji, na działanie algorytmu genetycznego. Zwiększenie liczby osobników może prowadzić do większej różnorodności rozwiązań i lepszego przeszukiwania przestrzeni rozwiązań, ale także do wzrostu kosztów obliczeniowych. W eksperymencie porównano działanie algorytmu dla populacji o wielkości 100, 200 i 400 osobników, przy stałych pozostałych parametrach.

[55]:
```
test_parameter('pop_size', [100, 200, 400], base_params)
```

```
Testuję pop_size = 100
Testuję pop_size = 200
Testuję pop_size = 400
```



Większa liczebność populacji (pop_size = 400) prowadzi do szybszej i skuteczniejszej poprawy najlepszych rozwiązań w kolejnych pokoleniach.

Populacje o wielkości 100 i 200 osobników również poprawiają wyniki, ale wolniej i zatrzymują się na gorszych lokalnych minimach.

Przy większej liczebności populacji algorytm znajduje krótsze trasy, co pokazuje korzyść z większej różnorodności rozwiązań, jednak należy pamiętać o zwiększonym czasie obliczeń.

Wizualizacja postępu algorytmu genetycznego dla pojedynczego przebiegu

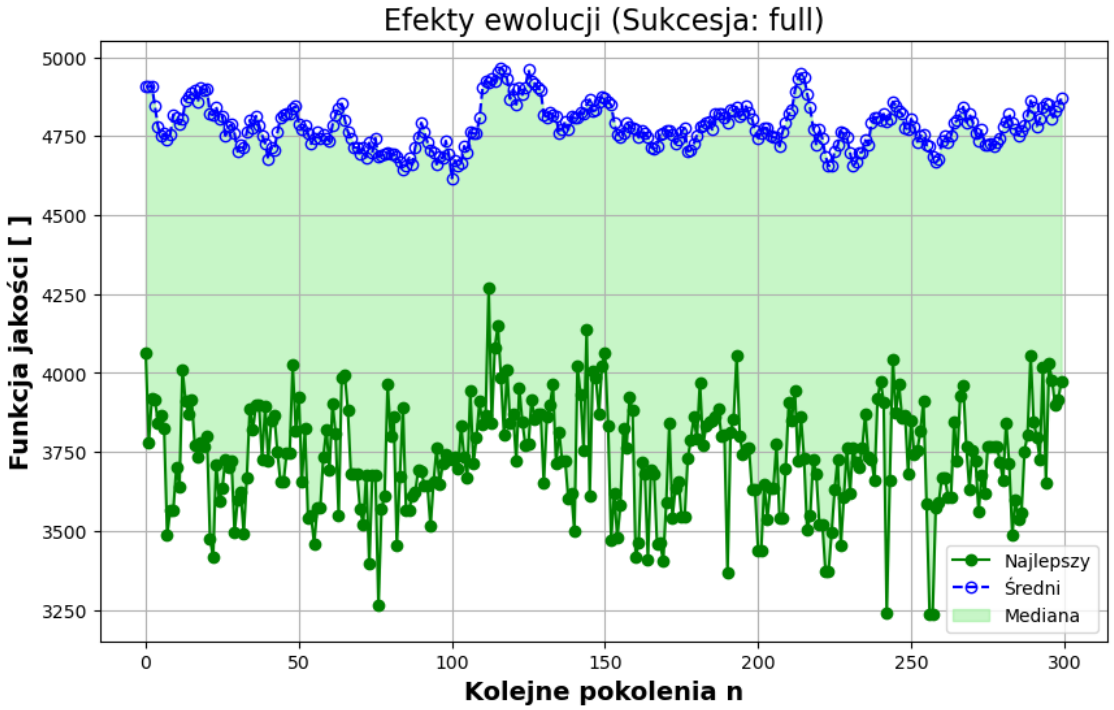
Aby dokładniej prześledzić przebieg ewolucji populacji w algorytmie genetycznym, przygotowano funkcję plot_single_evolution. Funkcja ta umożliwia jednoczesne przedstawienie zmian najlepszego i średniego wyniku w kolejnych pokoleniach. Dodatkowo wizualizowana jest przestrzeń pomiędzy tymi wartościami, co pozwala ocenić stabilność oraz zróżnicowanie jakości rozwiązań w czasie działania algorytmu.

[58]:
```
def plot_single_evolution(best_scores, avg_scores, title):
    generations = np.arange(len(best_scores))

    plt.figure(figsize=(10, 6))

    plt.plot(generations, best_scores, 'g-', marker='o', label='Najlepszy')
    plt.plot(generations, avg_scores, 'b--', marker='o', fillstyle='none',␣
↪label='Średni')
```

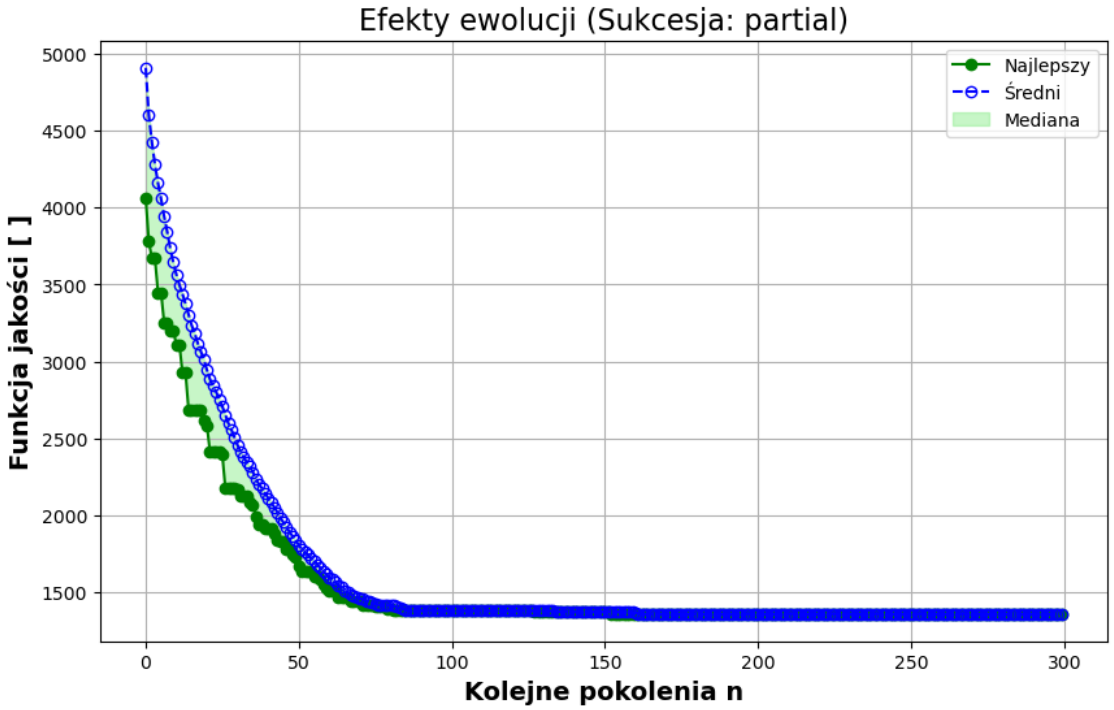
```
    plt.fill_between(generations, best_scores, avg_scores, color='lightgreen',␣
 ↪alpha=0.5, label='Mediana')

    plt.title(f'Efekty ewolucji ({title})', fontsize=16)
    plt.xlabel('Kolejne pokolenia n', fontsize=14, fontweight='bold')
    plt.ylabel('Funkcja jakości [ ]', fontsize=14, fontweight='bold')
    plt.legend()
    plt.grid(True)
    plt.show()
```

[61]:
```
common_params = {
    "n_cities": 30,
    "pop_size": 200,
    "cx_prob": 0.9,
    "mut_prob": 0.05,
    "n_generations": 300,
    "selection_method": "roulette"
}

for succession_type in ["full", "partial", "elitism"]:
    print(f"Liczymy dla sukcesji: {succession_type}")
    np.random.seed(42)
    cities, best_route, best_scores, avg_scores, worst_scores,␣
 ↪best_routes_per_generation, best_generation, best_score = genetic_algorithm(
        **common_params,
        succession_type=succession_type
    )
    plot_single_evolution(best_scores, avg_scores, title=f"Sukcesja:␣
 ↪{succession_type}")
```

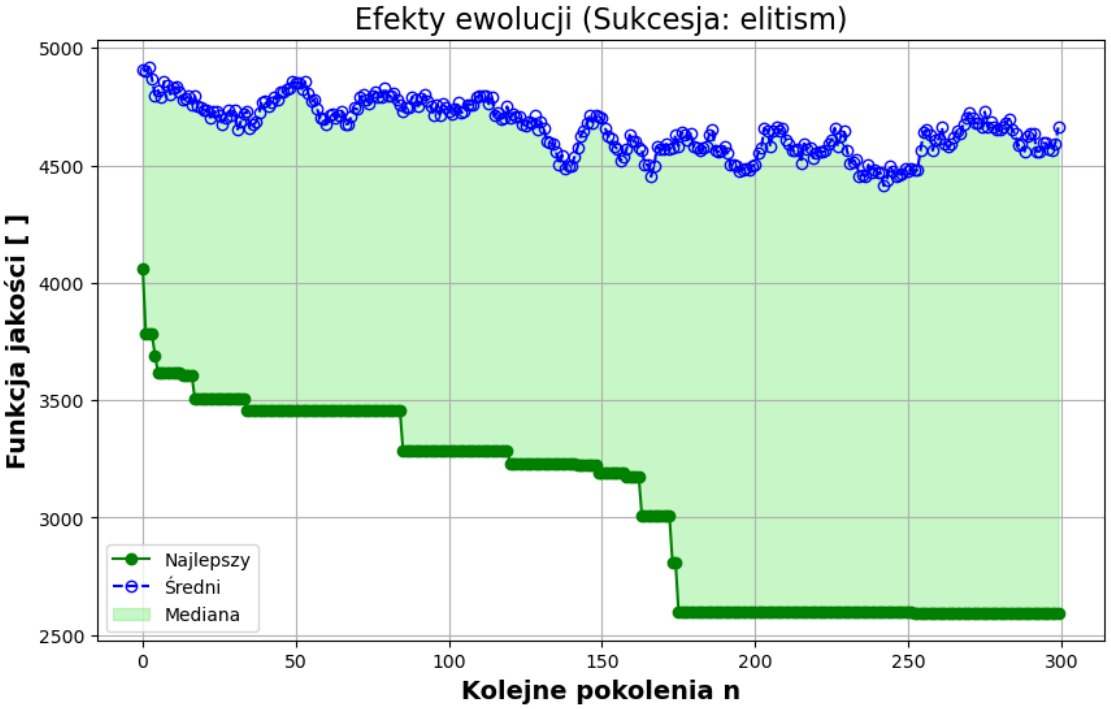
```
Liczymy dla sukcesji: full
```



Liczymy dla sukcesji: partial



```
Liczymy dla sukcesji: elitism
```



Sukcesja full

Widzimy bardzo duże wahania jakości rozwiązań w kolejnych pokoleniach.

Algorytm nie wykazuje wyraźnego trendu poprawy — populacja jest zmieniana całkowicie w każdym kroku, co zwiększa eksplorację, ale utrudnia stabilną optymalizację.

Duża rozpiętość między najlepszym a średnim osobnikiem wskazuje na brak utrzymywania dobrych rozwiązań.

Sukcesja partial

Funkcja jakości szybko się poprawia — krzywe szybko schodzą w dół i stabilizują się.

Różnica między najlepszym a średnim wynikiem maleje, co świadczy o zbliżaniu się całej populacji do dobrego rozwiązania.

Sukcesja elitism

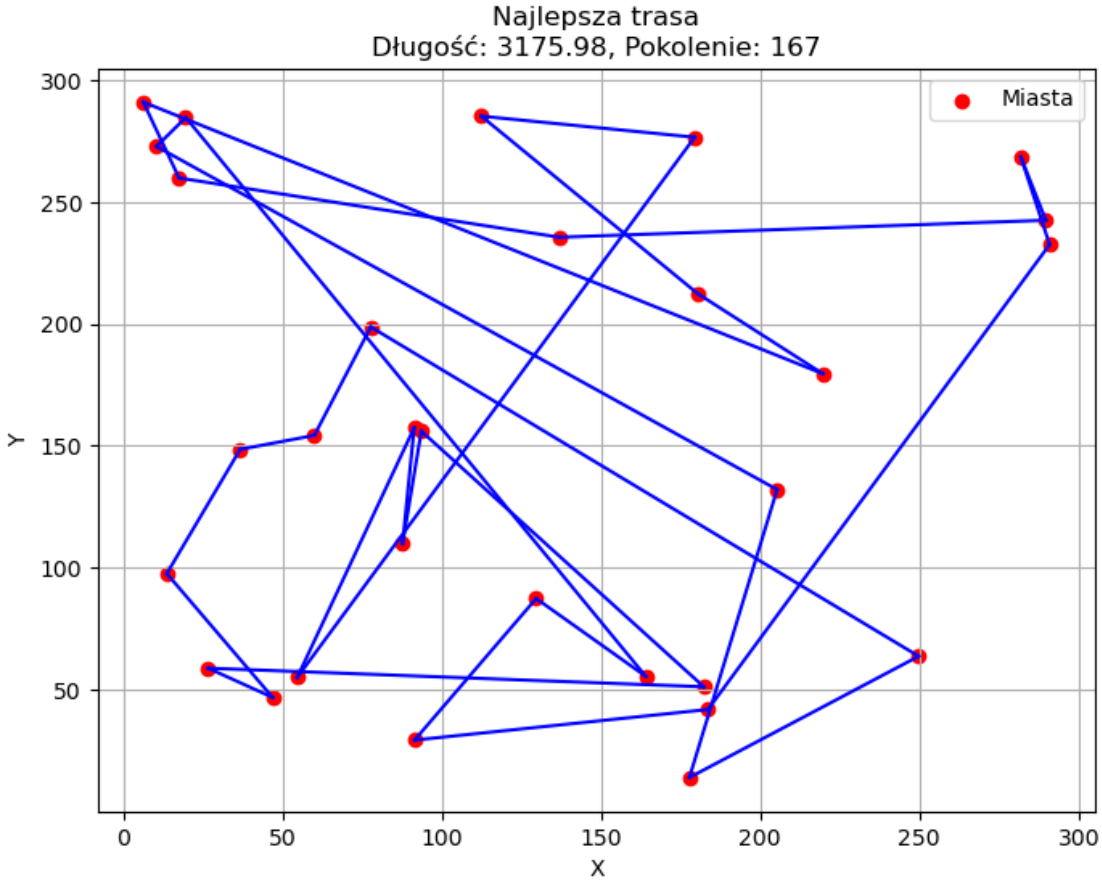
Najlepsze rozwiązanie poprawia się skokowo (stopniowo) — elitarny osobnik jest zawsze zachowany.

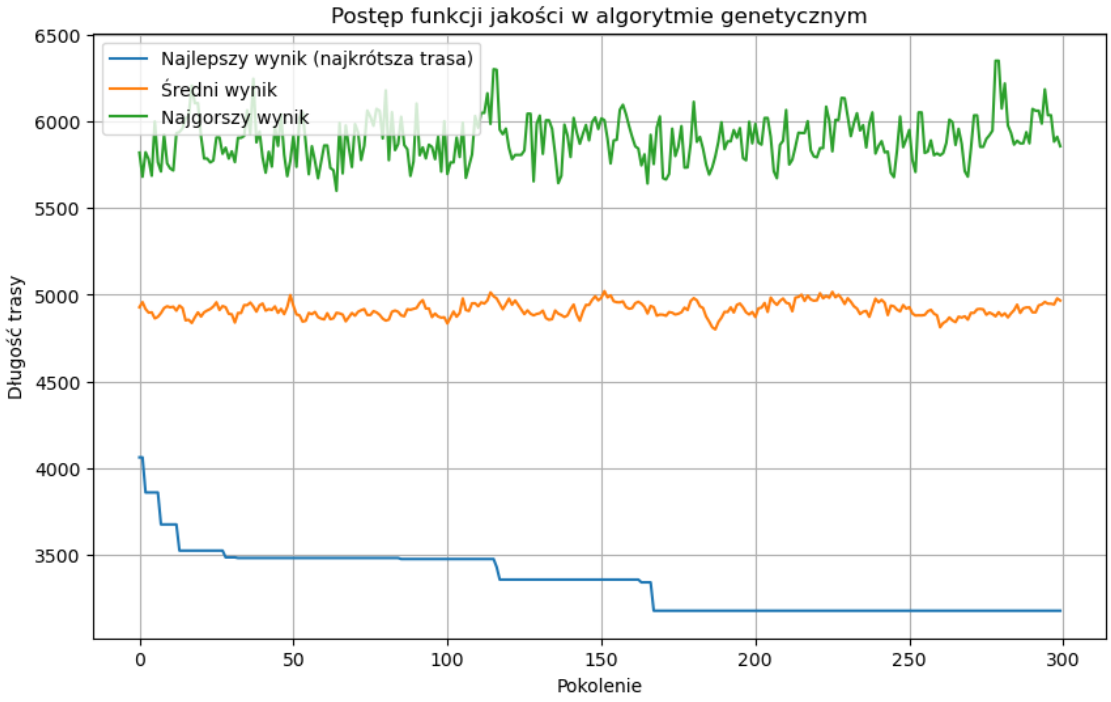
Średni wynik populacji pozostaje wysoki, a różnica między średnim a najlepszym wynikiem jest znaczna.

Sukcesja elitarna skutecznie chroni dobre rozwiązania, ale populacja jako całość ewoluuje wolniej.

1.0.6 Udowodnienie optymalizacji rozwiązania poprzez modyfikacje parametrów

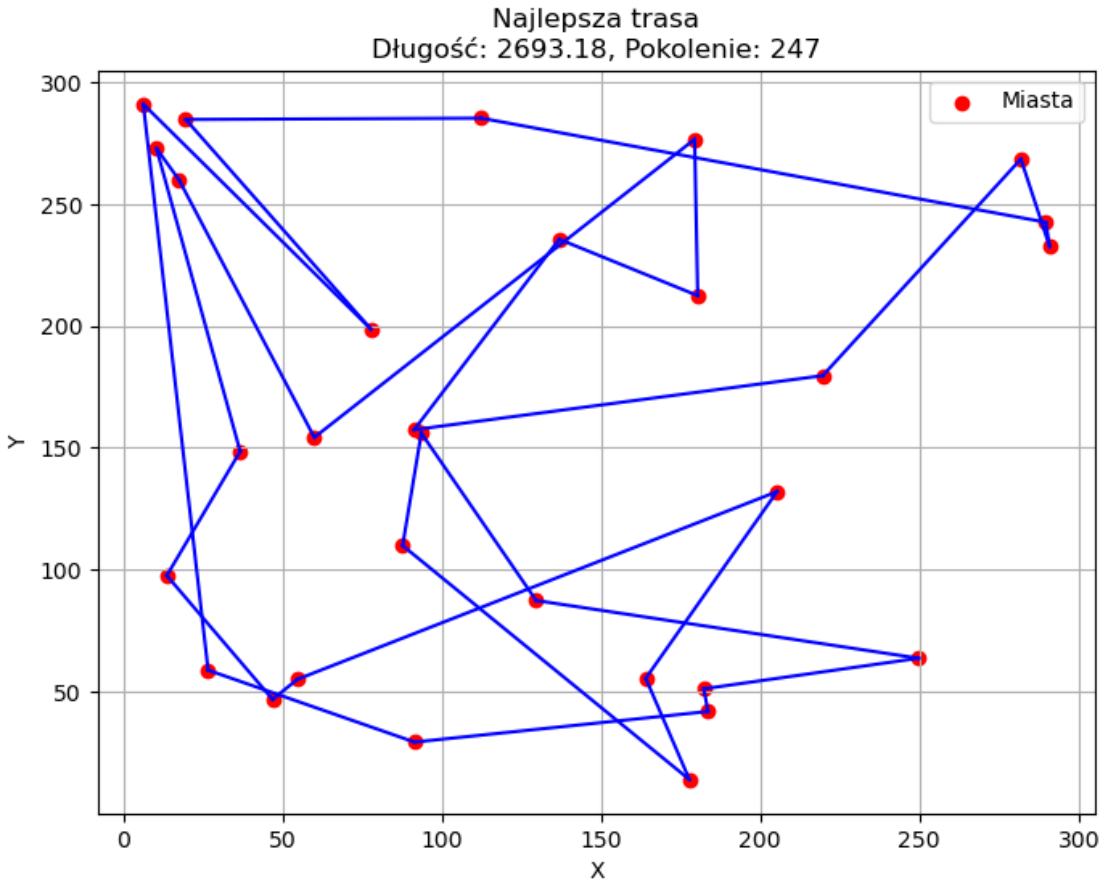
[70]:
```
np.random.seed(42)
cities, best_route, best_scores, avg_scores, worst_scores,␣
 ↪best_routes_per_generation, best_generation, best_score = genetic_algorithm(
    n_cities=30,
    pop_size=100,
    cx_prob=0.9,
    mut_prob=0.05,
    n_generations=300,
    selection_method="ranking",
    succession_type="elitism"
)
plot_route(cities, best_route, generation=best_generation, distance=best_score)
plot_scores(best_scores, avg_scores, worst_scores)
```

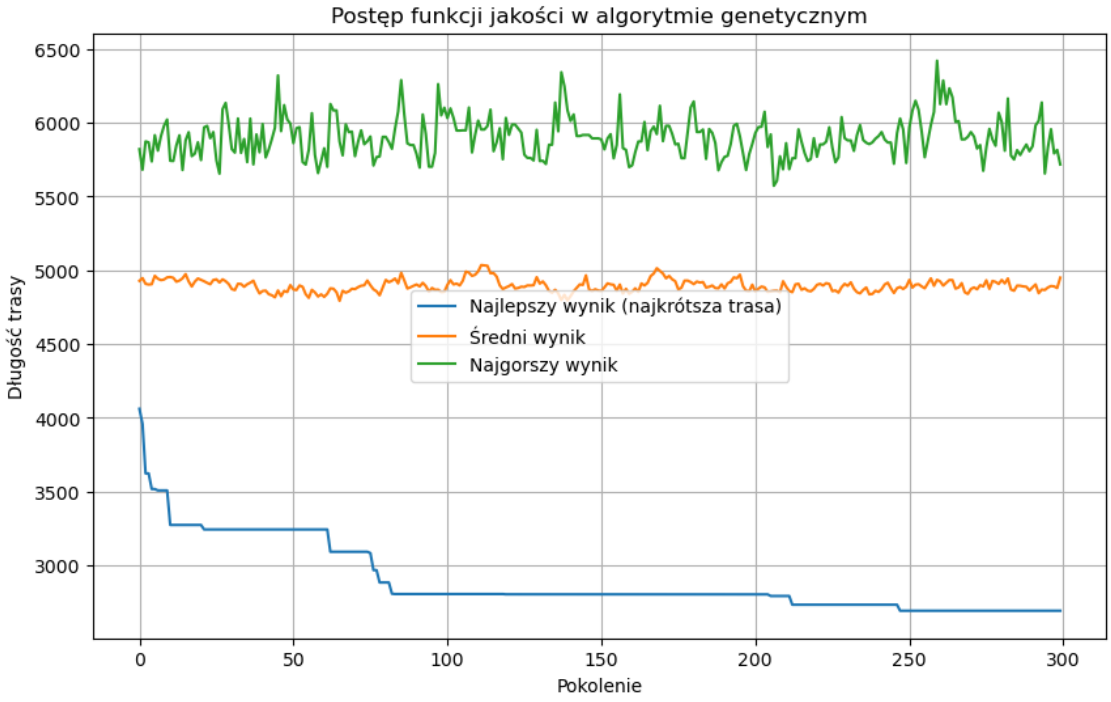




Dla przyjętych parametrów (n_cities=30, pop_size=100, cx_prob=0.9, mut_prob=0.05, n_generations=300, selekcja ranking, sukcesja elitism), algorytm genetyczny znalazł rozwiązanie o długości trasy 3175.98 w 167 pokoleniu.

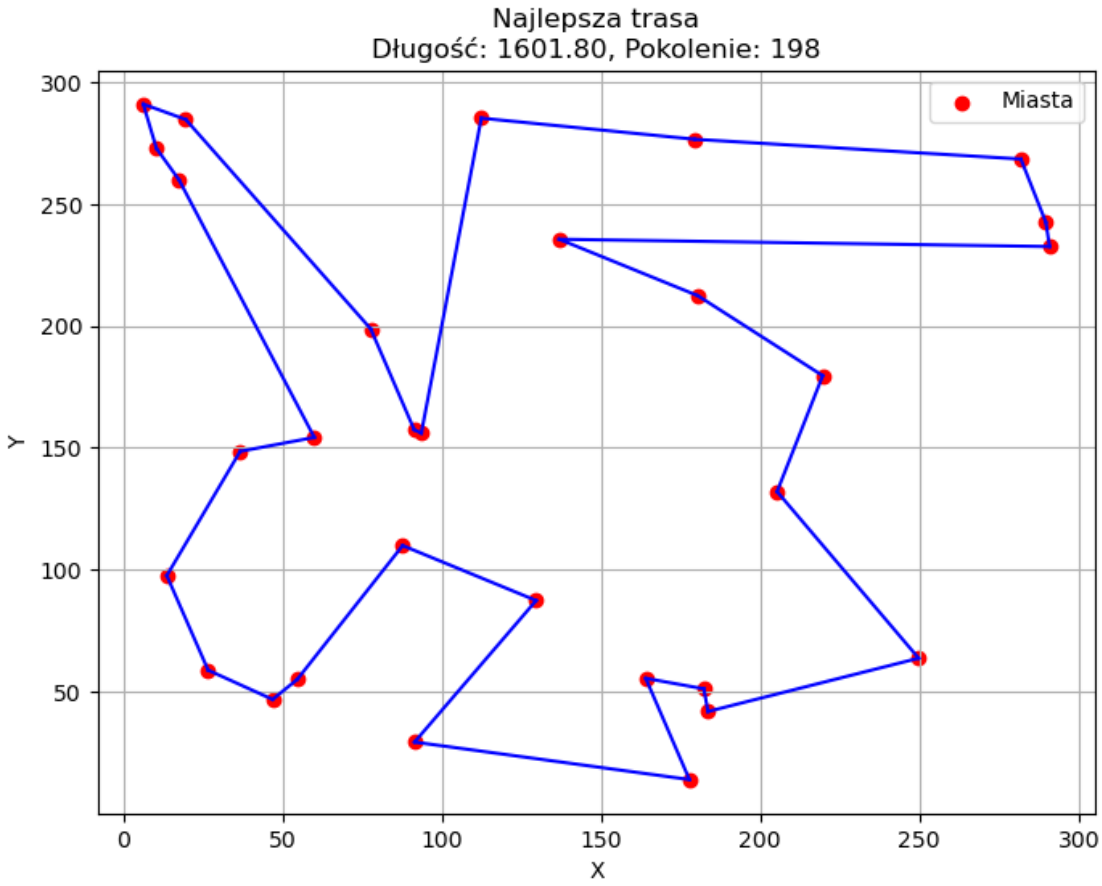
[71]:
```
np.random.seed(42)
cities, best_route, best_scores, avg_scores, worst_scores,␣
 ↪best_routes_per_generation, best_generation, best_score = genetic_algorithm(
    n_cities=30,
    pop_size=100,
    cx_prob=0.9,
    mut_prob=0.2,
    n_generations=300,
    selection_method="ranking",
    succession_type="elitism"
)
plot_route(cities, best_route, generation=best_generation, distance=best_score)
plot_scores(best_scores, avg_scores, worst_scores)
```

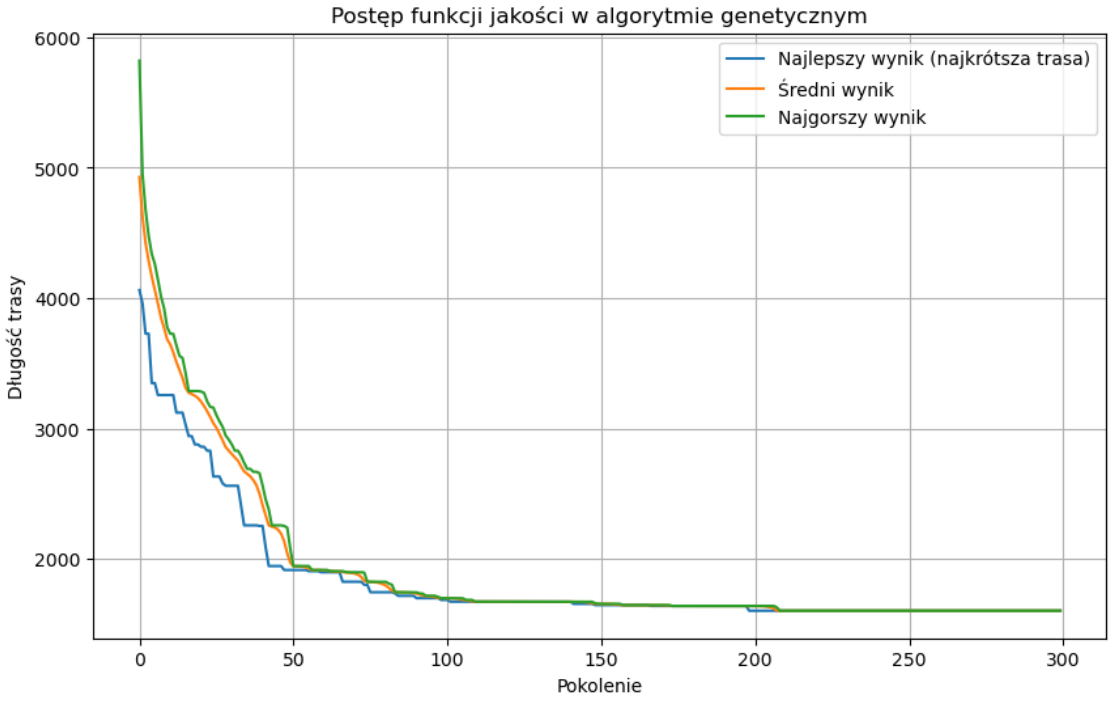

Najlepsza trasa
Długość: 2693.18, Pokolenie: 247
Miasta
300
250
200
150
100
50
Y
0
50
100
150
200
250
300
X



Po zwiększeniu prawdopodobieństwa mutacji do 0.2, algorytm znalazł lepsze rozwiązanie o długości 2693.18 w 247 pokoleniu. Wyższe prawdopodobieństwo mutacji wprowadziło więcej różnorodności do populacji, co pomogło uniknąć szybkiego utknięcia w lokalnym minimum. Proces ewolucji trwał dłużej, zanim osiągnięto najlepszy wynik, ale finalna trasa jest znacznie krótsza i bardziej logiczna.

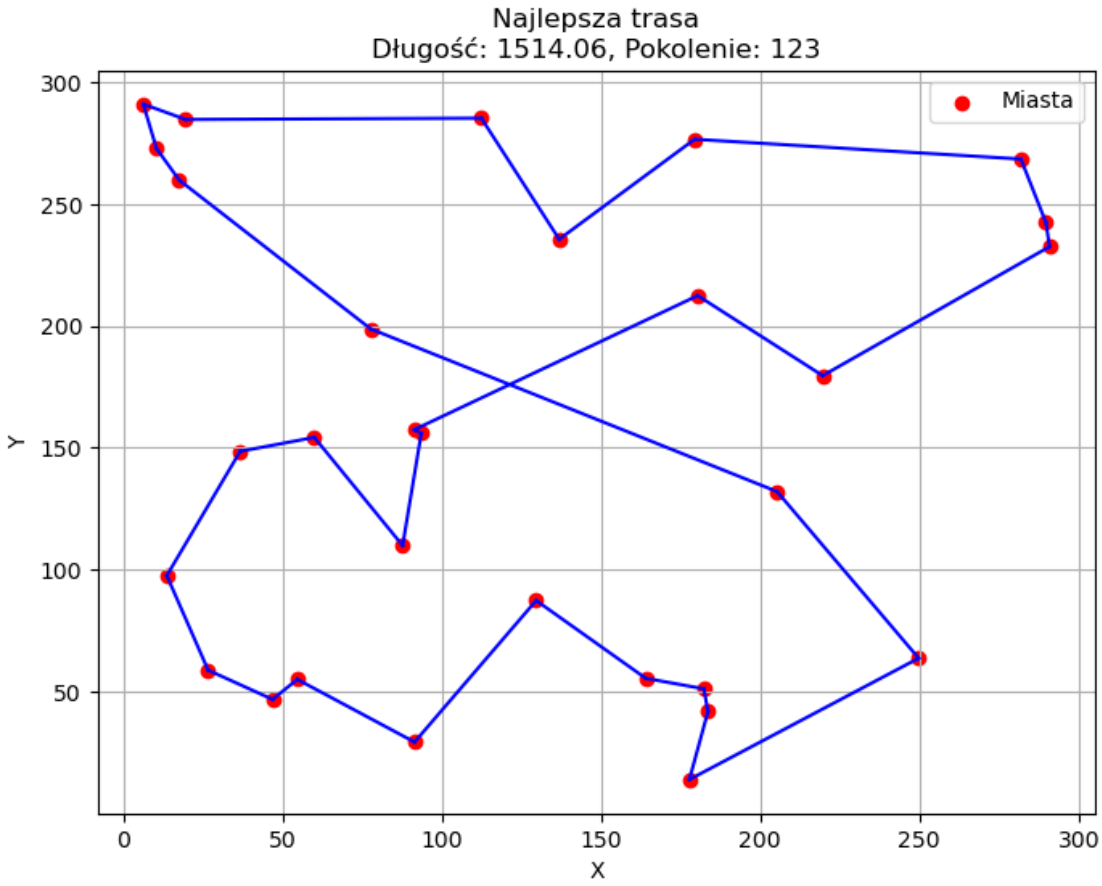
[79]:
```
np.random.seed(42)
cities, best_route, best_scores, avg_scores, worst_scores,␣
 ↪best_routes_per_generation, best_generation, best_score = genetic_algorithm(
    n_cities=30,
    pop_size=100,
    cx_prob=0.9,
    mut_prob=0.2,
    n_generations=300,
    selection_method="ranking",
    succession_type="partial"
)
plot_route(cities, best_route, generation=best_generation, distance=best_score)
plot_scores(best_scores, avg_scores, worst_scores)
```

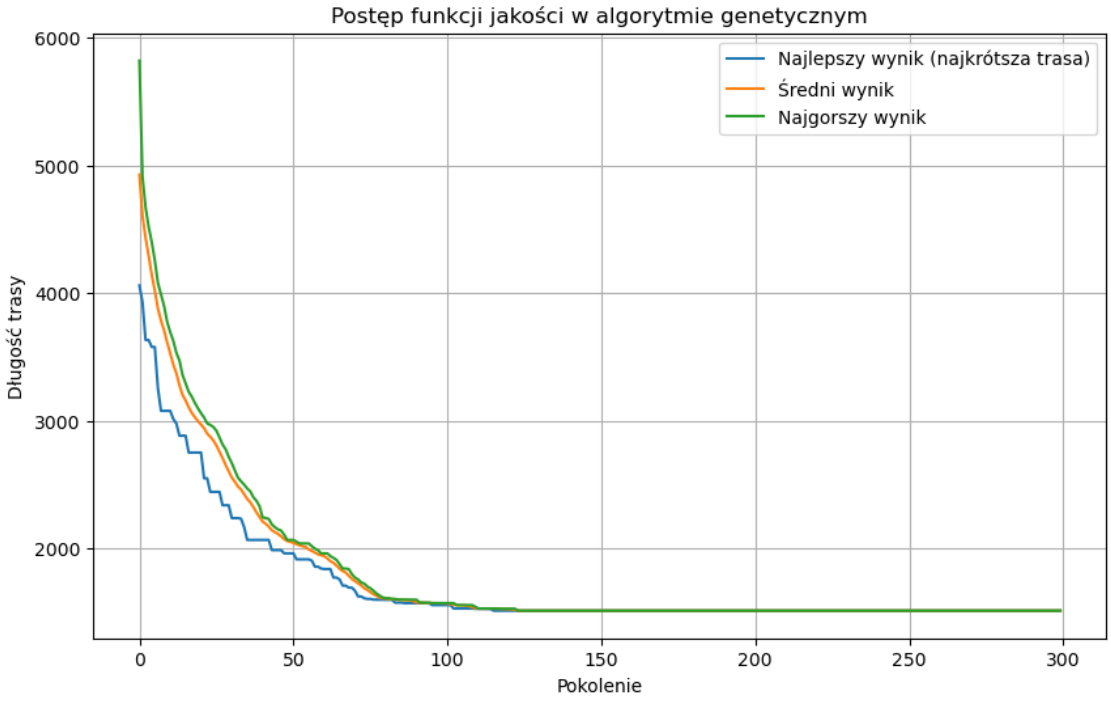

Najlepsza trasa
Długość: 1601.80, Pokolenie: 198
Miasta
Y
X
0
50
100
150
200
250
300



Po zastosowaniu częściowego zastępowania (succession_type="partial"), algorytm uzyskał znacznie lepszy wynik – trasę o długości 1601.80 osiągniętą w 198 pokoleniu. Częściowa sukcesja pozwala zachować część najlepszych osobników z poprzednich pokoleń, co sprzyja stopniowej poprawie jakości rozwiązań bez utraty różnorodności. W efekcie proces ewolucji jest bardziej stabilny i efektywny w znajdowaniu krótszych tras. Zmiana typu sukcesji miała kluczowy wpływ na poprawę końcowego wyniku i przyspieszenie osiągnięcia wysokiej jakości rozwiązania.

[82]:
```
np.random.seed(42)
cities, best_route, best_scores, avg_scores, worst_scores,␣
 ↪best_routes_per_generation, best_generation, best_score = genetic_algorithm(
    n_cities=30,
    pop_size=100,
    cx_prob=0.9,
    mut_prob=0.2,
    n_generations=300,
    selection_method="roulette",
    succession_type="partial"
)
plot_route(cities, best_route, generation=best_generation, distance=best_score)
plot_scores(best_scores, avg_scores, worst_scores)
```


Najlepsza trasa
Długość: 1514.06, Pokolenie: 123
Miasta
Y
X
0
50
100
150
200
250
300



Po zastosowaniu selekcji ruletkowej (selection_method="roulette") algorytm znalazł jeszcze lepsze rozwiązanie – trasę o długości 1514.06 już w 123 pokoleniu. Selekcja ruletkowa sprzyja większej różnorodności populacji, ponieważ osobniki są wybierane proporcjonalnie do ich jakości, a nie tylko na podstawie pozycji rankingowej. Dzięki temu algorytm szybciej wychodził z lokalnych minimów i skuteczniej poprawiał jakość trasy. Zmiana metody selekcji na ruletkową okazała się bardzo korzystna w kontekście szybkości zbieżności i jakości uzyskanego rozwiązania.

Ogólne wnioski:

Przeprowadzone eksperymenty potwierdziły, że odpowiedni dobór parametrów algorytmu genetycznego ma kluczowe znaczenie dla jakości uzyskiwanych rozwiązań. Zmieniając wartości takich parametrów jak prawdopodobieństwo mutacji, metoda selekcji czy sposób sukcesji, można znacząco poprawić wynik końcowy – zarówno pod względem długości najlepszej trasy, jak i szybkości osiągnięcia dobrych rozwiązań. Dzięki optymalizacji ustawień algorytmu udało się skrócić długość trasy nawet o kilkadziesiąt procent w stosunku do początkowych wyników. Badania te jednoznacznie dowodzą, że dostrajanie parametrów jest nieodłącznym elementem skutecznego zastosowania algorytmu genetycznego w praktyce.

[ ]: