04.2022

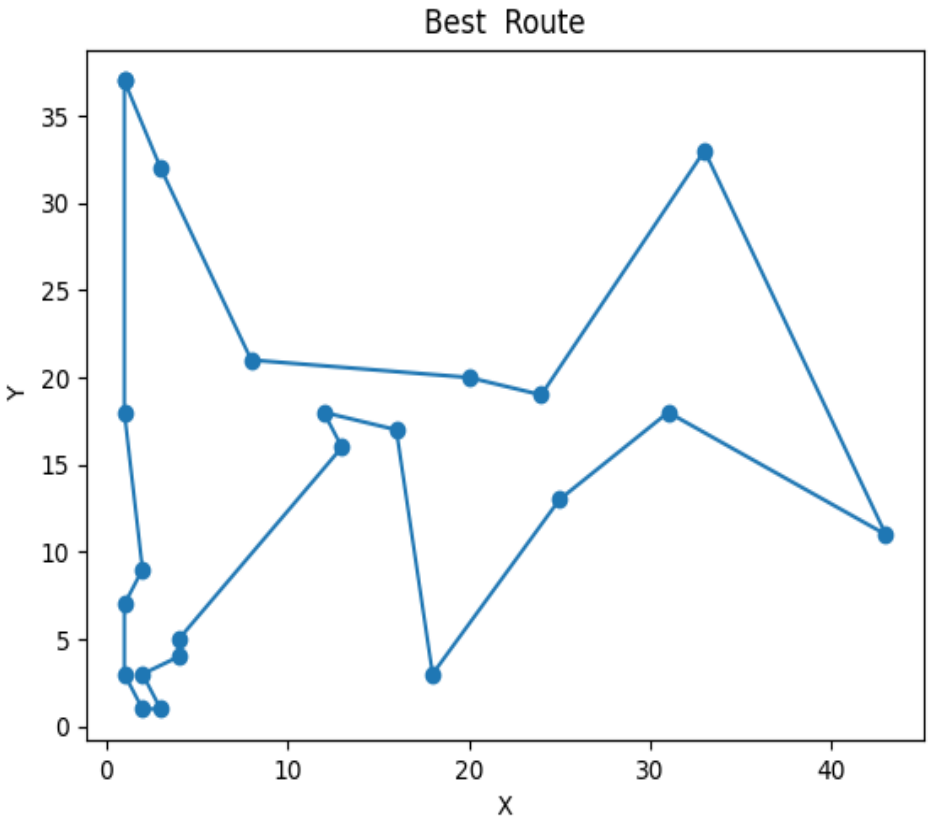
Zadanie projektowe nr 1

Problem komiwojażera

Michał Bidziński

1. Opis problemu

W problemie komiwojażera (Travelling salesman problem), kupiec musi odwiedzić wszystkie miasta (zaczynając i kończąc w tym samym mieście), tak aby przebyty dystans był jak najkrótszy. Problem planowania trasy staje się skomplikowany, gdy zwiększymy ilość miast oraz ich położenie pomiędzy sobą. Jeśli obliczana jest każda możliwa ścieżka, wymaga ona dużych zasobów obliczeniowych. W celu znalezienia optymalnej trasy zastosujemy różnorodne algorytmy m.in. algorytm genetyczny, roje cząsteczek oraz Ant Colony Optimizer (algorytm „mrówkowy”).



Przykładowy graf przedstawiający trasę kupca

2. Algorytmy

2.1. Algorytm genetyczny

Do algorytmu genetycznego wykorzystamy bibliotekę *pygad*.
Funkcje pomocnicze:

```
cities = pd.read_csv('cities.csv', index_col=0, header=0)
# liczba "miast"
N_CITIES = len(cities.index)


# oblicz odległość pomiędzy dwoma miastami
def calculate_distance(X, Y):
    return np.sqrt((X[1] - Y[1]) ** 2 + (X[0] - Y[0]) ** 2)


# Oblicz odleglosc pomiedzy wszystkimi miastami
# Stworz tablice 2-wymiarowa z n-miastami gdzie kazda tablica ma n-
elementow ( odleglosci od poszczegolnych miast)
# odejmujemy "-1" aby indeksować tablice miast od 0
def calculate_all_distances():
    array2d = [[0 for i in range(N_CITIES)] for y in
range(N_CITIES)]
    for i in range(1, N_CITIES + 1):

        for j in range(1, N_CITIES + 1):
            array2d[i - 1][j - 1] =
calculate_distance(cities.loc[i], cities.loc[j])
    return array2d


allDistances = calculate_all_distances()
```

Funkcja fitness nr 1:

```
# zdefiniowanie funkcji fitness
# odejmujemy "-1" aby indeksować tablice  od 0
# im wyższy fitness tym lepiej
def fitness_func(solution, solution_idx):
    dist = 0

    for i in range(N_CITIES - 1):
        x = int(solution[i] - 1)
        y = int(solution[i + 1] - 1)
        dist += allDistances[x][y]
    first = int(solution[0] - 1)
    last = int(solution[N_CITIES - 1] - 1)
    dist += allDistances[first][last]
    return 1 / dist
```

Funkcja fitness nr 2:

```
def fitness_func(solution, solution_idx):

    return 1 / sum(
        [allDistances[int(solution[i % N_CITIES] - 1), int(solution[(i + 1)
% N_CITIES] - 1)] for i in range(N_CITIES)])
```

W przypadku alg. genetycznego obie funkcje fitness działają podobnie. Do testów wykorzystamy funkcję pierwszą, gdyż jest bardziej czytelniejsza.

Pomocnicze funkcje do iteracji. Możemy wybrać parametr x, tak aby po x iteracjach bez zmian program się zatrzymał i zwrócił wynik.

```
def on_generation(g):
    print("Generation:", g.generations_completed, "\tDistance:",
          round(1 / g.last_generation_fitness[0], 4))

    # po "x" iteracjach bez zmian w wyliczanej najkrótszej odległośći
zatrzymujemy alogrytm.  "reach_criteria" w pygad
    x = 100
    if stop_criteria(g, x):
        return "stop"
```

funkcja stop_criteria porównuje bieżącą wartość fitness z „x” wcześniejszą wartością i jeżeli nie doszło do żadnych zmian – kończymy działanie programu.

```
def stop_criteria(g, x):
    if g.generations_completed > x:
        last_index = g.generations_completed - 1
        # bieżąca wartość fitness, którą porównujemy z "x" wcześniejsza
        # wartościa fitness i sprawdzamy, czy doszło
        # do jakiś zmian
        last_fitness = g.best_solutions_fitness[last_index]

        # wartosć fitness od której liczymy ilość iteracji bez żadnych
        zmian
        x_back_fitness = g.best_solutions_fitness[last_index - x]
        if math.isclose(last_fitness, x_back_fitness, abs_tol=0.00003):
            return True
    return False
```

funkcja do wypisania przebytego dystansu:

```
# wypisanie ostatecznego dystansu jaki musiał przebyc kurier
def print_distance(g):
    dist = fitness_func(g, 1)
    print("Final distance: ", 1 / dist)
```

parametry funkcji fitness:

- num_generations
  liczbę populacji możemy ustawić na wysoką np. 5000, ponieważ posiadamy funkcję stop_criteria, która w razie „x” iteracji bez zmian zatrzyma funkcje. Zbyt niska liczba spowoduje, że moglibyśmy nie znaleźć rozwiązania
- sol_per_pop
  rozmiar populacji - zbyt wysoki wydłuża czas działania programu, zbyt niski rozmiar ma tendencję do wpadania w „minimum” lokalne funkcji.
- num_genes
  liczba genów – ustawiamy równą liczbie miast
- parent_selection_type
  Ustawiamy na rank, gdyż szybciej znajduje rozwiązanie. Można ustawić również na steady, gdyż z reguły znajduje lepsze rozwiązanie w dłuższym czasie.
- mutation_type
  Najlepiej sprawdził się typ mutacji „adaptive”. Mutacja „random” może powodować, że odrzucimy zbyt dużo rozwiązań, a przy tym mało rozwiązań złych.
- gene_space
  Nasza dziedzina to oczywiście wszystkie miasta, czyli liczby całkowite od 1 do liczby miast + 1.
- num_parents_mating
  Wyłaniamy 25% rodziców do „rozmnażania”
- keep_parents
  Zachowujemy 5% rodziców
- crossover_type
  Ustawiamy na „single_point”

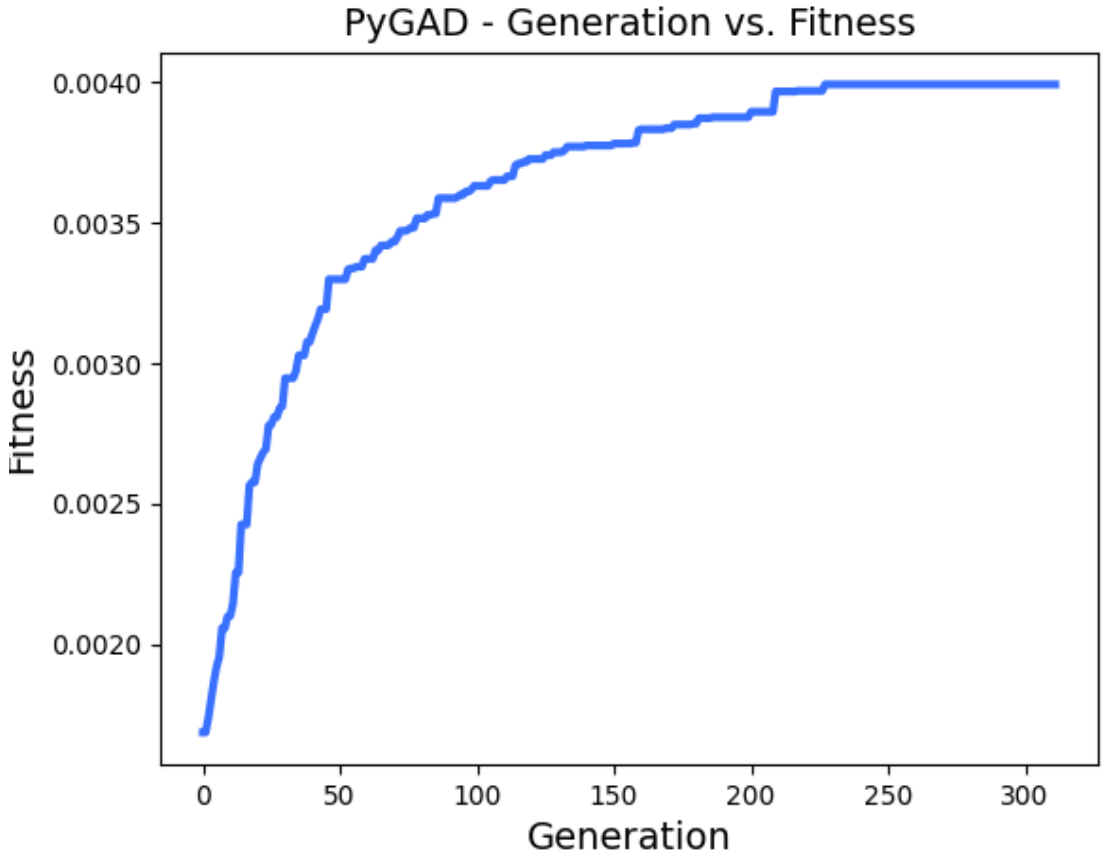
Całość parametrów wygląda następująco:

```
num_generations = 5000

sol_per_pop = 250

num_genes = N_CITIES

gene_space = range(1, N_CITIES + 1)

mating_percent = 25
num_parents_mating = math.ceil(sol_per_pop * mating_percent / 100)

keep_percent = 5
keep_parents = math.ceil(sol_per_pop * keep_percent / 100)

parent_selection_type = „rank”

crossover_type = „single_point”

mutation_type = „adaptive”

mutation_percent_genes = [20, 6]
```

Są to parametry, które pozwalają nam znaleźć rozwiązanie dla dużych jak i małych danych. Jeżeli chcemy porównać alg. genetyczny z innymi sposobami rozwiązania problemu komiwojażera, warto zmienić wartość num_generations na równą ilości iteracji jaką ustalimy dla innych algorytmów. Na czas programu największy wpływ ma jednak sol_per_pop czyli liczba chromosomów w populacji. Zmniejszenie tej liczby np. do 25 sprawi, że program zakończy się kilkukrotnie szybciej. Warto wziąć to pod uwagę, w przypadku małej liczby miast.

Test dla 40 miast o różnych współrzędnych

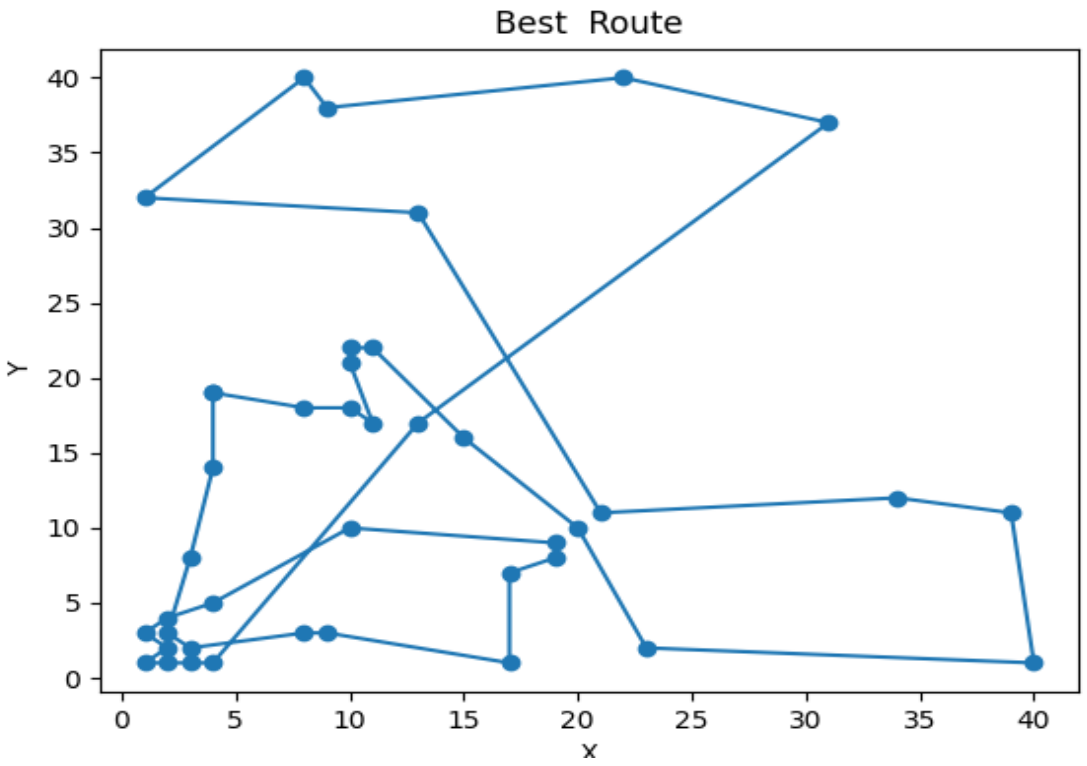
```
Czas działania programu 28.672194957733154
Best Route:
 [16. 32.  9.  7.  6. 29.  5. 28.  1. 27. 25. 31. 24. 33. 10. 14. 26. 13.
  2.  4.  3. 11. 19. 15. 20. 12. 22. 36. 35. 34. 17. 18.  8. 21. 38. 39.
 40. 30. 23. 37. 16.]

Fitness value of the best solution = 0.003992866577961838
Final distance:  250.44663538706337
```



Jak widzimy na wykresie, już po około 230 iteracjach wynik się nie zmieniał. Program kontynuował działanie przez 100 iteracji, gdyż na tyle była ustawiona wartość „x” w funkcji stop_criteria.
Czas działania programu to 28 sekund. Dla 40 miast można ustawić niższą liczbę populacji np. 25, aby uzyskać podobny wynik w czasie 4 sekund.

```
Generation: 397     Distance: 262.408
Czas działania programu 3.8999550342559814
Best Route:
 [34. 36. 35. 11. 19. 15. 20. 12. 22.  6.  4. 26. 13.  2. 27.  1. 28.  5.
 29.  3. 38. 39. 40. 30. 33. 25.  7.  9. 16. 32. 37. 17. 18. 21.  8. 23.
 24. 31. 10. 14. 34.]

Fitness value of the best solution = 0.003810859952528983
```

liczba populacji = 25, czas = 3.89s



4 testy dla 40 miast o różnych współrzędnych

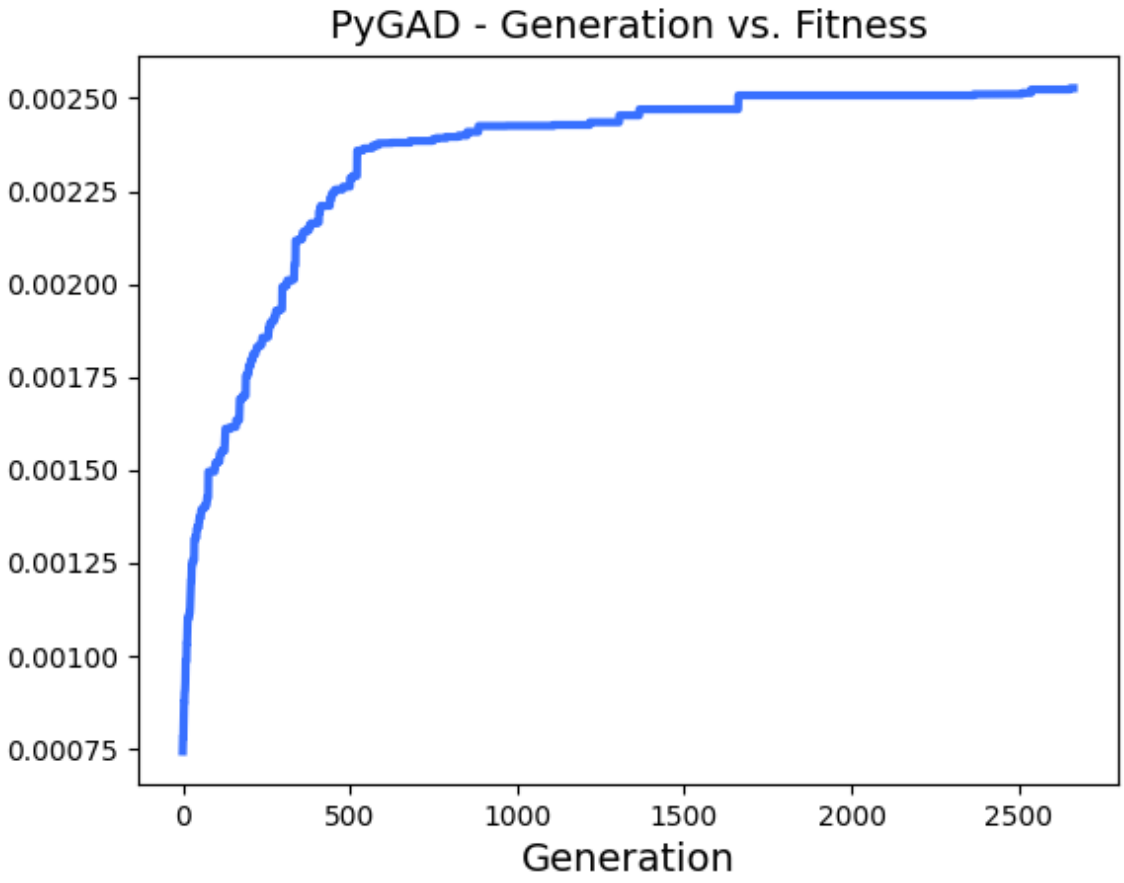
```
Czas działania programu 16.893880605697632
Best Route:
 [19. 15. 20. 12. 18. 21.  8. 23. 24. 31. 33.  3.  5.  1. 28. 13. 26. 30.
 40. 39. 38.  9.  7.  6. 29.  4.  2. 27. 10. 14. 25. 17. 37. 16. 32. 34.
 35. 36. 22. 11. 19.]

Fitness value of the best solution = 0.0038989681740550447
Final distance:  256.4781130182886
```

```
Czas działania programu 21.00284767150879
Best Route:
 [ 8. 21. 18. 17. 22. 11. 19. 15. 20. 12. 39. 40. 38. 23. 24.  6.  3. 29.
  4.  5.  1. 28. 13. 26.  2. 27. 14. 10. 25. 16. 34. 36. 35.  9.  7. 32.
 37. 31. 33. 30.  8.]

Fitness value of the best solution = 0.004063460645106695
Final distance:  246.09565277917017
```

```
Czas działania programu 24.742741346359253
Best Route:
 [12. 20. 15. 19. 11.  4.  2. 26. 13. 27. 33. 31. 14. 10. 28.  1.  5. 29.
 32. 16. 37. 17. 25.  3.  6.  7.  9. 34. 18. 23.  8. 21. 38. 39. 40. 30.
 24. 36. 35. 22. 12.]

Fitness value of the best solution = 0.003557045060353364
Final distance:  281.13222718091123
```

```
Czas działania programu 27.144319772720337
Best Route:
 [36. 22. 12. 20. 15. 19. 11.  6.  3. 25. 23.  8. 21. 38. 39. 40. 30. 33.
 29.  4.  5. 28.  1. 13. 26.  2. 27. 10. 14. 31. 24. 18. 17. 37. 16. 32.
  7.  9. 34. 35. 36.]

Fitness value of the best solution = 0.004219495665243604
Final distance:  236.99514807826375
```

Dla liczby populacji równej 250 i num_generations równym 5000 najniższy dystans jaki uzyskaliśmy to 236 z czasem 27 sekund. Udało się nam również znaleźć rozwiązanie w 16 sekund z dystansem 256. Średni czas dla 4 iteracji to 22 sekund z dystansem 254,75. Trzeba jednak pamiętać, że funkcja stop_criteria zatrzymywała działanie programu po 100 iteracjach bez zmian. **Zwiększmy liczbę miast do 80, a funkcje stop_criteria ustawmy na 1000 iteracji bez zmian.**



Widzimy, że funkcja fitness jest coraz wyższa, co skutkuje mniejszym dystansem, natomiast czas programu wydłuża się kilkukrotnie przy czym po 500 iteracjach wynik jest już satysfakcjonujący.

Spróbujmy teraz przetestować różne parametry algorytmu genetycznego

Na początku zacznijmy od **liczby populacji**

Liczba populacji	Dystans	Czas[s]	Fitness value	Generacja rozwiązania
50	312	5.44	0.0032	246
100	278	11.71	0.0036	279
200	253	15.26	0.0039	204
500	232	54.98	0.0043	271

Parametry*: num_generations = 500, stop_criteria = 100, 40 miast.*

Szansa mutacji

W wybranym przez nas typie mutacji (adaptive), procent mutacji ustalamy jako [x,y] gdzie x to procent mutacji rozwiązań gorszych od średniej, a y procentu mutacji dla rozwiązań lepszych od średniej. x powinien być większy od y.

Szansa mutacji[x,y]	Dystans	Czas[s]	Fitness value	Generacja rozwiązania
[5,2]	449	9.37	0.0022	306
[10,4]	368	17.71	0.0027	364
[15,4]	338	27.26	0.0029	479
[20,5]	353	41.88	0.0028	543

Parametry*: num_generations = 5000, stop_criteria = 100, 65 miast, liczba populacji = 100.*

2.2. Algorytm PSO

Paczka *pyswarms* nie obsługuje przypadków, w których dziedzina optymalizowanej funkcji to liczby całkowite, w związku z tym użyjemy biblioteki *scikit-opt*.

Użyjemy tych samych funkcji pomocniczych, oraz przetestujemy dwie funkcje fitness.

```
# liczba "miast"
N_CITIES = len(cities.index)


# oblicz odległość pomiędzy dwoma miastami
def calculate_distance(X, Y):
    return np.sqrt((X[1] - Y[1]) ** 2 + (X[0] - Y[0]) ** 2)


# Oblicz odleglos pomiedzy wszystkimi miastami
# Stworz tablice 2-wymiarowa z n-miastami gdzie kazda tablica ma n-
elementow ( odleglosci od poszczegolnych miast)
# odejmujemy "-1" aby indeksować tablice miast od 0

def calculate_all_distances():
    array2d = [[0 for _ in range(N_CITIES)] for _ in range(N_CITIES)]
    for i in range(1, N_CITIES + 1):

        for j in range(1, N_CITIES + 1):
            array2d[i - 1][j - 1] = calculate_distance(cities.loc[i],
cities.loc[j])
    return array2d


allDistances = np.array(calculate_all_distances())

# 1 funkcja fitness
def fitness_func(solution):
    return sum([allDistances[solution[i % N_CITIES], solution[(i + 1) %
N_CITIES]] for i in range(N_CITIES)])


# druga funkcja fitness
def fitness_func(solution):
    dist = 0

    for i in range(N_CITIES - 1):
        x = int(solution[i] - 1)
        y = int(solution[i + 1] - 1)
        dist += allDistances[x][y]
    first = int(solution[0] - 1)
    last = int(solution[N_CITIES - 1] - 1)
    dist += allDistances[first][last]
    return dist
```

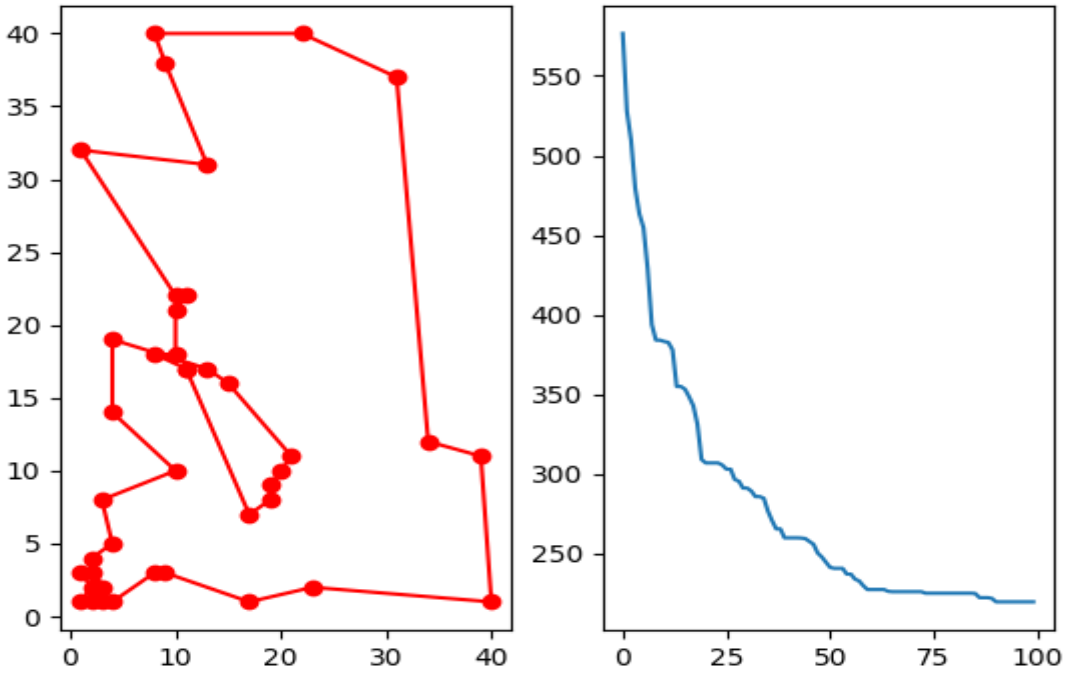
Parametry PSO

- n_dim
  liczbę wymiarów ustawimy równą liczbie miast
- size_pop
  liczba populacji
- max_iter
  liczba iteracji

```
pso_tsp = PSO_TSP(func=fitness_func, n_dim=N_CITIES, size_pop=250,
max_iter=100, w=0.8, c1=0.1, c2=0.1)

best_points, best_distance = pso_tsp.run()

print('Najlepszy dystans', best_distance)
```

Test PSO dla 40 miast o różnych współrzędnych

```
# 1 funkcja fitness
def fitness_func(solution):
    return sum([allDistances[solution[i % N_CITIES], solution[(i + 1) %
N_CITIES]] for i in range(N_CITIES)])
```

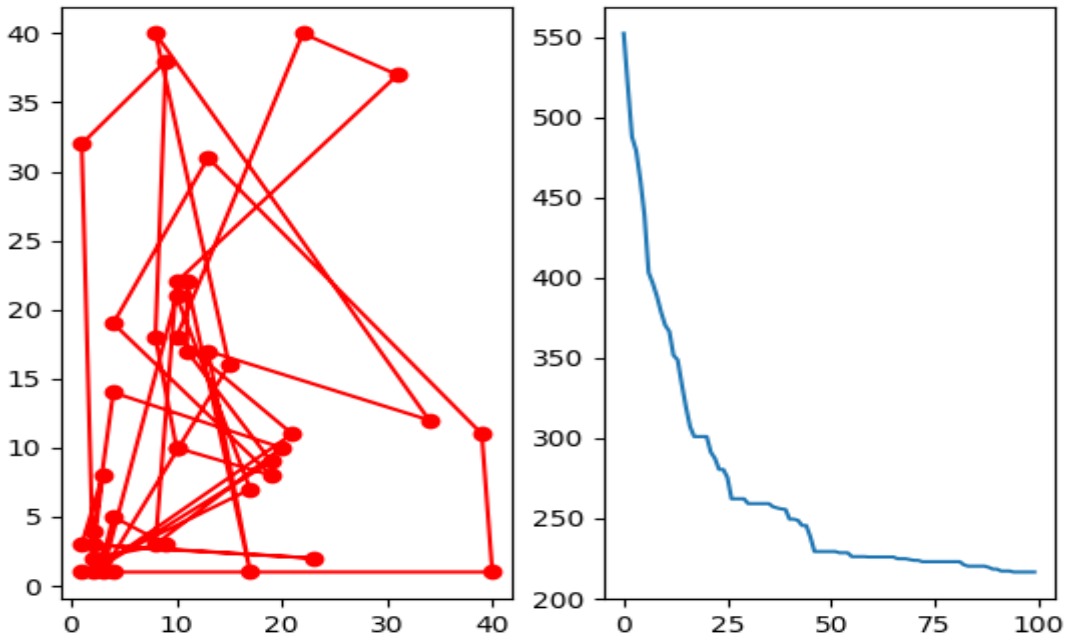


1 funkcja fitness

```
Czas działania programu 2.735250949859619
Najlepszy dystans [212.41727671]
```

PSO znajduje rozwiązanie szybciej, a także w przypadku naszych danych lepiej. W tylko 100 iteracjach dystans mieści się w przedziale od 210 do około 250. **Sprawdźmy, jak PSO poradzi sobie z szukaniem rozwiązania, gdy użyjemy tej samej funkcji, co w przypadku algorytmu genetycznego.**

```python
# druga funkcja fitness
def fitness_func(solution):
    dist = 0

    for i in range(N_CITIES - 1):
        x = int(solution[i] - 1)
        y = int(solution[i + 1] - 1)
        dist += allDistances[x][y]
    first = int(solution[0] - 1)
    last = int(solution[N_CITIES - 1] - 1)
    dist += allDistances[first][last]
    return dist
```




```
Czas działania programu 4.360092878341675
Najlepszy dystans [209.37409589]
```

PSO z użyciem tej same funkcji, co algorytm genetyczny również znajduje rozwiązanie szybciej. Funkcja 2 działa natomiast wolniej, niż funkcja 1, a w dodatku otrzymujemy mniej czytelny graf ścieżek. Dla algorytmu PSO najlepszym rozwiązaniem będzie **funkcja 1.**

Liczba iteracji	Liczba populacji	Czas[s]	Dystans
100	100	1.097	262
100	200	2.234	229
200	300	6.589	220
200	400	8.9	200
1000	10	1.16	214
2000	10	2.36	223

testy dla funkcji 1

Wniosek: Lepiej zwiększyć liczbę iteracji, niż liczbę populacji (dla opracowywanych danych).

2.3. Ant Colony Optimization

Algorytmy mrówkowe zostały zainspirowane obserwacją życia mrówek. Technika rozwiazywanie problemów za pomocą tego algorytmu sprawdza się bardzo dobrze w przypadku komiwojażera. Gdy mrówka odnajdzie dobrą (krótką) drogę, inne mrówki będą podążać tą właśnie drogą, a pozostałe drogi zostają odrzucane.

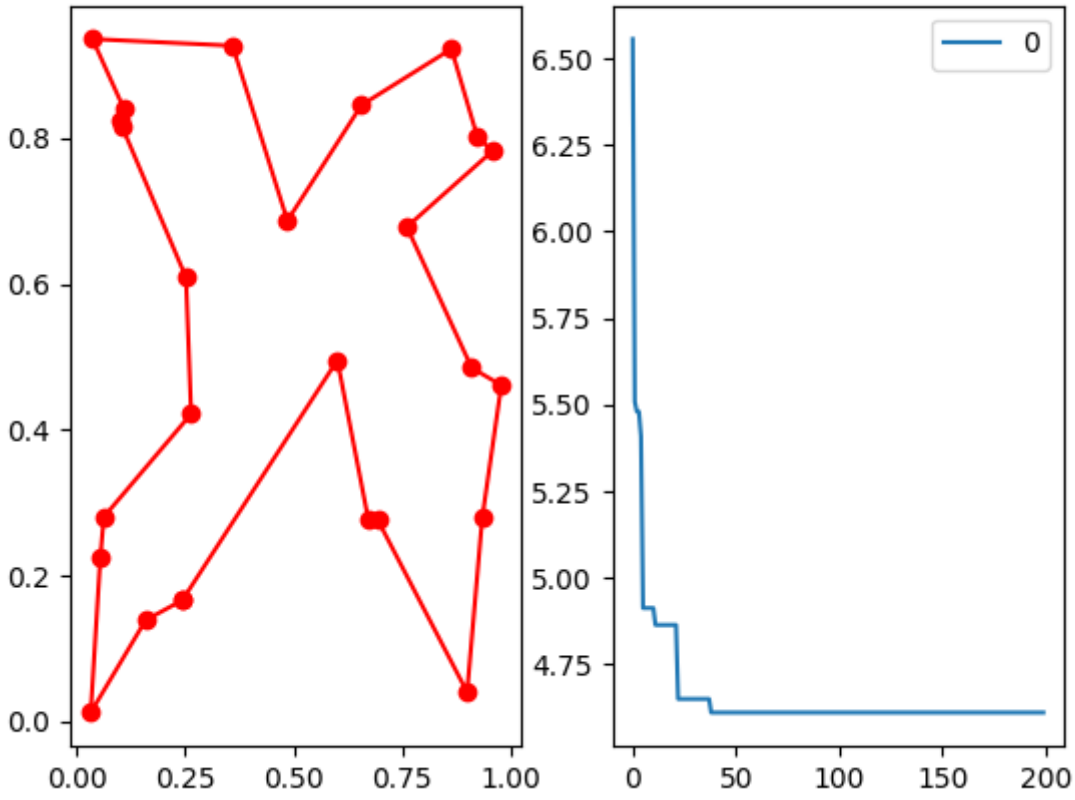
Ponownie wykorzystamy bibliotekę *scikit-opt* (https://github.com/guofei9987/scikit-opt/blob/master/sko/ACA.py) oraz wcześniej wykorzystane już funkcje.

```
def calculate_distance(X, Y):
    return np.sqrt((X[1] - Y[1]) ** 2 + (X[0] - Y[0]) ** 2)
def calculate_all_distances():
    array2d = [[0 for _ in range(N_CITIES)] for _ in range(N_CITIES)]
    for i in range(1, N_CITIES + 1):

        for j in range(1, N_CITIES + 1):
            array2d[i - 1][j - 1] = calculate_distance(cities.loc[i],
cities.loc[j])
    return array2d
allDistances = np.array(calculate_all_distances())

def fitness_func(solution):
```

```
    return sum([allDistances[solution[i % num_points], solution[(i + 1) %
num_points]] for i in range(num_points)])
```

Rozpoczniemy testy od 40 miast, 200 iteracji oraz wielkości populacji równej 50.

```
aca = ACA_TSP(func=fitness_func, n_dim=num_points,
              size_pop=50, max_iter=200,
              distance_matrix=allDistances)
start = time.time()
best_x, best_y = aca.run()
end = time.time()
czas = end - start

print("Distance", best_y)
print("Czas działania programu", czas)
```



Wynik programu:

```
Distance 204.97569242619036
Czas działania programu 23.83017921447754
```

Jak widzimy, algorytm „mrówkowy” **najlepiej** radzi sobie z szukaniem najkrótszej ścieżki. Niestety ma również swoje wady, ponieważ w porównaniu z innymi algorytmami, jest on najwolniejszy. Przy liczbie populacji równej 50 oraz 200 iteracjach czas wynosił 23 sekundy (dla porównania algorytm PSO przy takich parametrach znajdował rozwiązanie w 2-4sekundy)

Sprawdźmy wyniki ACO z innymi parametrami

Liczba populacji	Liczba iteracji	Czas[s]	Dystans
50	600	67.43s	204

Zwiększenie liczby iteracji nie daje nam lepszego wyniku, a zwiększa czas kilkukrotnie

50	50	6.23	211

Zmniejszenie liczby iteracji daje gorszy wynik, ale o wiele krótszy czas

25	200	12.6s	202

Zmniejszenie liczby populacji daje lepszy wynik i krótszy czas(dla danych 40 miast).

2	200	0.91	218

Oczywiście, nie możemy zmniejszać populacji drastycznie, jednakże nawet dla populacji równej 2 program znajduje wynik w 0.91sekundy przy dystansie 218, co i tak jest imponującym wynikiem

3. Podsumowanie

Dla 40 miast, każdy z algorytmów był w stanie znaleźć ścieżkę. Zdecydowanym zwycięzca tego zestawienia jest algorytm „mrówkowy”, który znajdował rozwiązania najlepsze w najkrótszym czasie. Najsłabiej wypadł algorytm genetyczny, który w podobnym czasie co PSO i ACO znajdował zdecydowanie dłuższe trasy.