Problem komiwojażera

Katarzyna Gruszczyńska

Luty 2022

Spis treści

1 Opis problemu

Problem komiwojażera opisuje się następująco: dla zadanej liczby miast, komiwojażer musi odwiedzić każde miasto dokładnie jeden raz wracając do punktu początkowego. Zadane są koszty przejazdu pomiędzy każdą parą miast. Rozwiązanie problemu polega na znalezieniu takiej drogi dla komiwojażera, aby każde miasto zostało odwiedzone dokładnie jeden raz i całkowity koszt przejazdu był jak najmniejszy.

Problem komiwojażera należy do klasy problemów NP-trudnych.

2 Algorytm genetyczny i implementacja

Do rozwiązania problemu zastosowano algorytm genetyczny.

Zadeklarowano 100 miast (genów), tj. N = 100. Populacja początkowa liczy 50 osobników otrzymanych losowo. Tabela kosztów podróży jest generowaną losowo macierzą symetryczną. Funkcją dopasowania jest koszt podróży. Wybór osobników odbywa się poprzez selekcję turniejową binarną:
The term "binary tournament" refers to the size of two in a tournament, which is the simplest form of tournament selection. Binary tournament selection (BTS) starts by selecting two individuals at random. Then, fitness values of these individuals are evaluated. The one having more satisfactory fitness is then chosen [1]

Zastosowano krzyżowanie jednopunktowe z odwróceniem kolejności genów, tzn. geny podlegające krzyżowaniu są dokładane do drugiego osobnika dziedziczącego w odwrotnej kolejności. W literaturze można znaleźć, że takie podejście przynosi lepsze rezultaty, np:
"Evolutionary reversal operation can improve the local search ability of genetic algorithms. This paper introduces evolutionary reversal operation after selection, cross and mutation operation. The reverse algorithm is as follows: first generate 2 random integers n1 and n2 randomly n1,n2 [1,10] to determine 2 positions, and then change the position of the two cities in the TSP solution. If a=2, b=5, the previous tour route is 1 2 5 10 8 6 3 7 4 9, after the reverse operation, the tour route is 1 5 2 10 8 6 3 7 4 9" [2].

Ponadto, przyjęto ze 1 na 1000 genów mutuje. Metodą mutacji jest zamiana miejscami genów/miast.

2.1 Kroki algorytmu

Poniżej przedstawiono najważniejsze kroki algorytmu:

- generowanie tabeli kosztów podróży:
  - wartości kosztów: 10$ - 99$, zapisane w losowej macierzy symetrycznej,
- generowanie populacji początkowej dla zadanej liczby osobników i genów (miast),
- obliczanie funkcji dopasowania (Fitness function), kosztu podróży - liczony jest dla pojedynczego osobnika,
- liczone są koszty dla wszystkich osobników w populacji i znajdowany jest minimalny koszt,
  - koszty podroży dla wszystkich osobników w początkowej populacji / kolejnych pokoleniach
  - znajdowany jest koszt minimalny/najkrótsza trasa w początkowej populacji / kolejnych pokoleniach
- stosowana jest selekcja turniejowa w celu wybrania populacji rodziców,
- tworzone jest nowe pokolenie, z puli rodziców po kolei brane są pary do krzyżowania,
  - następuje krzyżowanie osobników - jednopunktowe, z odwróceniem kolejności genów,
  - wybierany jest losowy punkt krzyżowania,
  - aby rozwiązanie było poprawne miasta nie mogą się powtarzać, więc usuwane są powtarzające się, które obecne są u drugiego rodzica,
  - zwracane są dzieci czyli nowe pokolenie,

[1] https://www.hindawi.com/journals/mpe/2016/3672758/
[2] https://aip.scitation.org/doi/abs/10.1063/1.5039131

- mutacja genów osobnika/ów
  - przyjęto, że mutuje 1 na 1000 genów - "mutation rate",
  - metoda mutacji - wzajemna wymiana (wybranie dwóch miast i zamienienie ich ze sobą),
  - do mutacji wybierany jest 1 osobnik na n-osobników, wg "mutation rate" i wylosowana para jego genów,
  - następuje zamiana genów czyli miast miejscami,
- uruchomienie całego algorytmu dla zadanej liczby pokoleń
  - rysowanie wykresu: koszt podróży dla każdego pokolenia
  - prezentacja kosztu minimalnego/najtańszej trasy w ostatnim pokoleniu populacji

2.2 Implementacja

Do implementacji algorytmu wykorzystano język Python:

- Kod źródłowy znajduje w repozytorium Github:

  `https://github.com/kasia-gruszczynska/TSP-Komiwojazer`

- Poniżej zamieszczono listing całego programu.

Dodano komentarze opisujące co dzieje się w funkcjach i najważniejszych fragmentach kodu.

```
# Feb 2022
# Katarzyna Gruszczynska
# UJ FAIS IGK
#
# TSP - Problem komiwojazera
#
import sys
import numpy as np
import random
import matplotlib.pyplot as plt

# ile miast, default N = 100
N = 100

# generowanie tabeli kosztow podrozy
# wartosci kosztow 10$ - 99$
# losowa macierz symetryczna
def generateCostTable(N):
    b = np.random.randint(10,99,size=(N,N),dtype=int)
    b_symm = (b + b.T)/2
    return b_symm

# generowanie osobnikow poczatkowych, default = 50
def generatePopulation(ile_osobnikow):
    population = []
    i = 0
    while i < ile_osobnikow:
        # print("+ nowy osobnik", i)
        population.append(random.sample(range(0, N, 1), N))
        i += 1
    print("Poczatkowa populacja osobnik w: ", len(population))
    # print(population)
    return population

# funkcja dopasowania, fitness function - koszt podrozy
# dla pojedynczego osobnika, n - numer osobnika
def fnDopasowania(population, costTable, n):
    cost = 0
    i = 0
    for i in range(N-1):
        # print(population[n][i], "->", population[n][i+1])
```

```
        cost += costTable[population[n][i], population[n][i+1]]
        # print(costTable[population[n][i], population[n][i+1]])
    print("Koszt podrozy: ", cost)
    return cost

# policz koszty dla wszystkich osobnikow w populacji
# znajdz minimalny
def minKosztPodrozy(population, costTable):
    minKoszt = 0
    kosztyOsobnikow = []
    n = 0
    # print("len(population) ----> ", len(population))
    for n in range(0, len(population)):
        kosztyOsobnikow.append(fnDopasowania(population, costTable, n))
        # print("Koszt dla osobnika", n ,":", kosztyOsobnikow[n])
    # print("Koszty osobnikow", kosztyOsobnikow)
    minKoszt = min(kosztyOsobnikow)
    print("Koszt minimalny/najkrotsza trasa", minKoszt)
    return minKoszt

# selekcja turniejowa
# Binary tournament selection (BTS)
def binTounamentSelection(population, costTable):
    # do turnieju 1:1 wylosowac osobnikow
    pair = random.sample(range(0, len(population)), 2)
    # print("para", pair)
    # print("population", population)

    if fnDopasowania(population, costTable, pair[0]) < fnDopasowania(population, costTable, pair
    [1]):
        rodzic = population[pair[0]].copy()
        # print("rodzic to osobnik nr", pair[0], population[pair[0]])
        # print("osobnik usuniety nr", pair[1], population[pair[1]])
        del population[pair[1]]
    else:
        rodzic = population[pair[1]].copy()
        # print("rodzic to osobnik nr", pair[1], population[pair[1]])
        # print("osobnik usuniety nr", pair[0], population[pair[0]])
        del population[pair[0]]

    print(population)

    return rodzic

# krzyzowanie osobnikow, jednopunktowe
def crossOver(rodzic1_val, rodzic2_val):
    # losowy punkt krzyzowania
    crossPoint = random.randint(1, N-2)
    rodzic1 = list(rodzic1_val)
    rodzic2 = list(rodzic2_val)
    rodzic1_orig = list(rodzic1)
    rodzic2_orig = list(rodzic2)

    dzieci = []
    # aby rozwiazanie bylo valid nie moga sie miasta powtarzac
    # usunac te wylosowane
    cr = crossPoint
    end = N - 1 - crossPoint
    i=0
    for i in range(end):
        rodzic2.remove(rodzic1[crossPoint+1+i])
        cr += 1
        i += 1
    temp = rodzic1[crossPoint+1:]
    temp.reverse()
    reversed = rodzic2 + temp
    dzieci.append(reversed)

```

```
    # print("dzieci", dzieci)

    rodzic1 = list(rodzic1_orig)
    rodzic2 = list(rodzic2_orig)

    cr = crossPoint
    end = N - 1 - crossPoint
    i=0
    for i in range(end):
        rodzic1.remove(rodzic2[crossPoint+1+i])
        cr += 1
        i += 1

    temp = rodzic2[crossPoint+1:]
    temp.reverse()
    reversed = rodzic1 + temp
    dzieci.append(reversed)

    # print("dzieci", dzieci)

    rodzic1 = list(rodzic1_orig)
    rodzic2 = list(rodzic2_orig)

    return dzieci

# tworz nowe pokolenie
# z puli rodzicow bierz pary
def krzyzowanieRodzicow(rodzice):
    newGeneration=[]

    for i in range(len(rodzice)-1):
        print("krzyzowanie")
        dzieci = crossOver(rodzice[i],rodzice[i+1])
        newGeneration.append(dzieci[0])
        newGeneration.append(dzieci[1])
    return newGeneration

# ewolucja, krzyzowanieRodzicow wywolaj dla populacji
def evolution(population, rodzice, costTable):
    # wybranie puli rodzicow
    while len(population) >= 2:
        # print("while len populacji:", len(population), population)
        rodzice.append(binTounamentSelection(population, costTable))

    print("pula rodzicow:", len(rodzice), rodzice)
    # print("ile osobnikow zostalo w populacji poczatkowej - wygina!:", len(population),
    population)

    newGeneration = krzyzowanieRodzicow(rodzice)

    # print("evolution ---> newGeneration", newGeneration)

    # zwroc dzieci czyli nowe pokolenie
    return newGeneration

def swapPositions(list, pos1, pos2):
    list[pos1], list[pos2] = list[pos2], list[pos1]
    return list

# mutacja, 1:1000 genow
# wzajemna wymiana (wybranie dw ch miast i zamienienie ich ze sob )
# do mutacji wybierac 1 osobnika na n-osobnikow, wg mutationRate
def mutation(newGeneration, mutationRate):
    i = 0
    while (mutationRate < len(newGeneration)):
        mutowany = random.randint(i, mutationRate)
        # do mutacji wylosowac pare genow osobnika
        genes = random.sample(range(0, N), 2)
```

```
        # print("geny do mutacji:", genes)
        # zamiana miast
        swapPositions(newGeneration[mutowany], genes[0], genes[1])
        i += mutationRate
        mutationRate += mutationRate
    # return mutated generation
    return newGeneration

def nextGeneration(population, costTable):
    rodzice = []
    newGeneration = []

    # print("pula rodzicow:", len(rodzice), rodzice)
    # print("len populacji:", len(population), population)

    # ewolucja przez krzyzowanie osobnikow - jednopunktowe
    newGeneration = evolution(population, rodzice, costTable)

    # mutacja - 1:1000 osobnik w
    # 1000 / 100 = 10 czyli co ~10ty osobnik mutuje
    # 1000 / N = mutationRate
    mutationRate = int(1000 / N)

    if (mutationRate < len(newGeneration)):
        mutation(newGeneration, mutationRate)
        print("zmutowana generacja", newGeneration)
    else:
        print("za duzy mutationRate!")

    # print("nextGeneration ----> newGeneration:", newGeneration)

    return newGeneration

# run geneticAlgorithm dla x pokolen
def geneticAlgorithm(ile_osobnikow, generations):

    print("Liczba miast:", N)
    print("Liczba osobnikow poczatkowych:", ile_osobnikow)
    print("Liczba pokolen:", generations)
    # generacja tabeli kosztow podrozy
    costTable = []
    print("\n-------- Tabela kosztow podrozy (10$ - 99$) --------")
    costTable = generateCostTable(N)
    print(costTable,"\n")

    # generuj populacje poczatkowa
    population = generatePopulation(ile_osobnikow)
    print("Ilosc osobnikow w populacji:", len(population))

    # koszty podrozy dla wszystkich osobnikow w poczatkowej populacji
    # i koszt minimalny/najkrotsza trasa w poczatkowej populacji
    minKoszt = minKosztPodrozy(population, costTable)
    print("Koszt minimalny/najkrotsza trasa w poczatkowej populacji:", minKoszt)

    progress = []
    progress.append(minKoszt)

    print("Tworzenie nowego pokolenia")
    nowePokolenie = population.copy()
    print("nowePokolenie = population", nowePokolenie)

    for i in range(0, generations):
        print("pokolenie nr", i+1)
        nowePokolenie = nextGeneration(nowePokolenie, costTable)
        print("generations -----> nowePokolenie", nowePokolenie)
        minKoszt = minKosztPodrozy(nowePokolenie, costTable)
        print("Koszt minimalny/najkrotsza trasa dla pokolenia nr", i+1 ,":", minKoszt)
        progress.append(minKoszt)
```

```
244
245     # koszty podrozy dla wszystkich osobnikow w ostatnim pokoleniu populacji
246     # i koszt minimalny/najkrotsza trasa w ostatnim pokoleniu populacji
247     # minKoszt = minKosztPodrozy(population, costTable)
248     print("Koszt minimalny/najkrotsza trasa w ostatnim pokoleniu populacji:", minKoszt)
249
250     plt.plot(progress)
251     plt.title('Problem komiwojazera')
252     plt.ylabel('Koszt podrozy')
253     plt.xlabel('Pokolenie')
254     plt.savefig('komiwojazer_cost_per_generation.png')
255     plt.show()
256
257     return 0
258
259 # ------------------------------ execute genetic algorithm --------------------------------
260
261 # ile miast, constant, default=100
262 # ile osobnikow, default=50
263 # ile pokolen, default=500
264
265 # argumenty podane z command line'a
266 ile_osobnikow = int(sys.argv[1])
267 generations = int(sys.argv[2])
268
269 # run everything and draw a plot: Koszt podrozy per Generation
270 geneticAlgorithm(ile_osobnikow, generations)
271
272 #
```

Listing 1: Kod źródłowy pliku komiwojazer.py

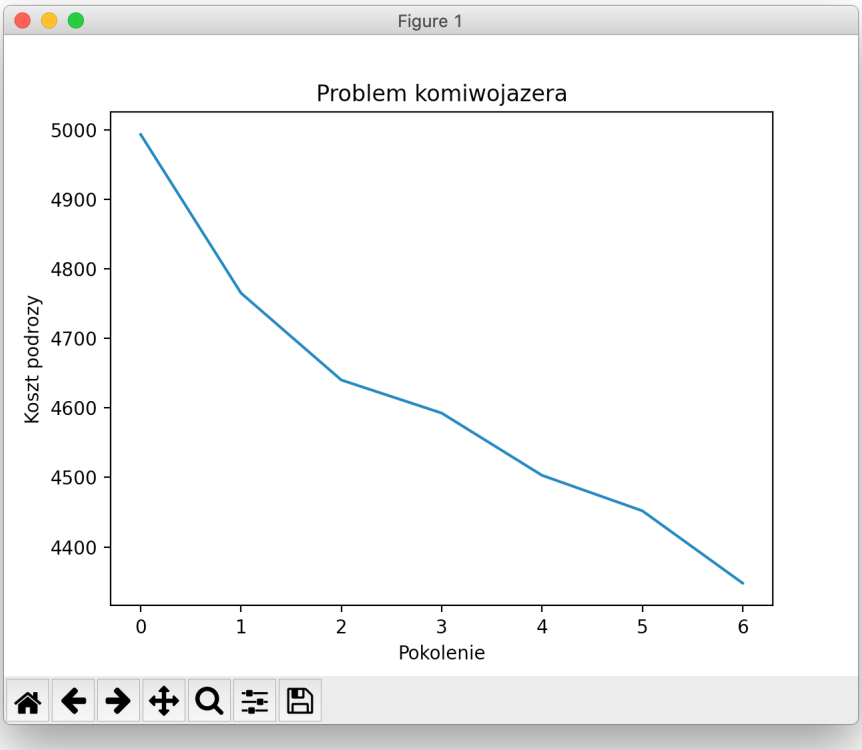
3 Przykład

3.1 Założenia

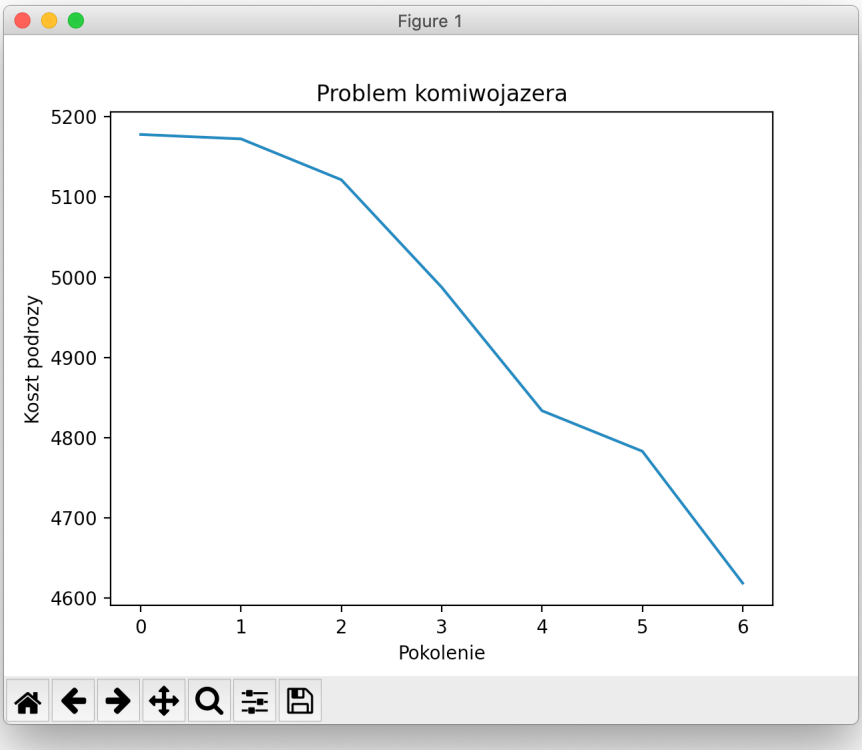
Dla zadanej liczby miast (N=100) oraz osobnikow poczatkowych (50), wykonanych zostanie x pokolen.
Podana max liczba pokolen, jest to threshold, stop condition algorytmu.
Na wyjściu otrzymujemy znalezioną ścieżkę o minimalnym koszcie oraz wykres koszt per pokolenie.
ile$_miast, default = 100ile_osobnikow, default = 50ilepokolen, default = 500$
ile$_miast = 10, N = 10ile_osobnikow = 6ile_generations = 600$

3.2 Przykładowe uruchomienie programu

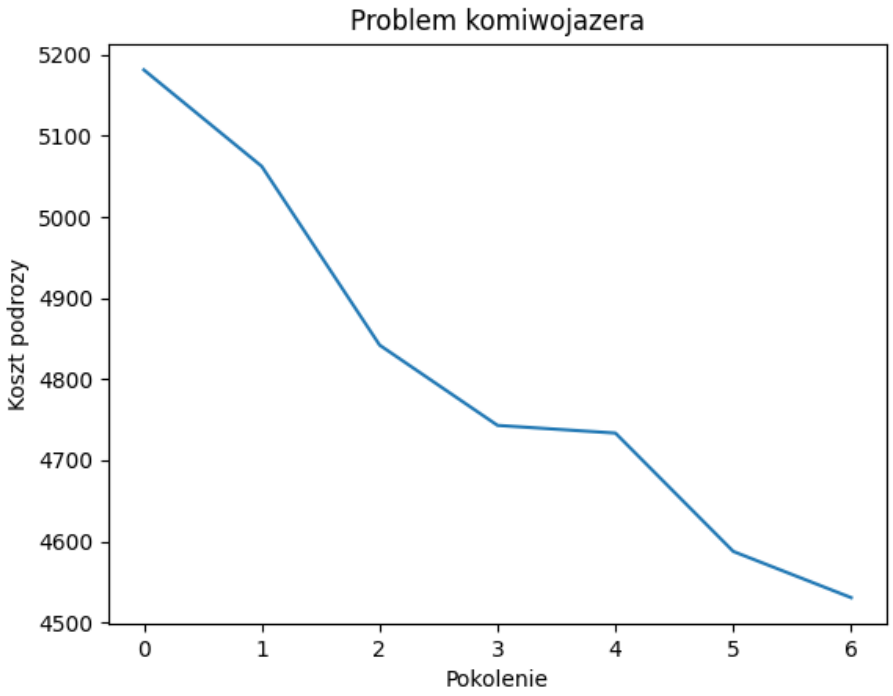
argumenty podane z command line'a ile$_osobnikowgenerationsliczbamiaststala$: $100, N = 100$
Poniżej, zrzuty ekranu oraz log wygenerowany w trakcie działania programu.
Wykonanie dla: Liczba miast: 100 Liczba osobnikow poczatkowych: 10 Liczba pokolen: 6



Rysunek 1: Przykładowy wykres: koszt per pokolenie. Źródło: własne



Rysunek 2: Przykładowy wykres: koszt per pokolenie. Źródło: własne



Rysunek 3: Przykładowy wykres: koszt per pokolenie. Źródło: własne

3.3 Log z wykonania

Poniżej zamieszczono przykładowe uruchomienia i wycinki z konsoli.

```
MBP-Kasia:komiwojazer_kod kasia$ python3 komiwojazer.py 10 6
Liczba miast: 100
Liczba osobnikow poczatkowych: 10
Liczba pokolen: 6

-------- Tabela kosztow podrozy (10$ - 99$) --------
[[79.  50.  31.5 ... 41.  43.  70. ]
 [50.  57.  89.5 ... 55.  82.5 68. ]
 [31.5 89.5 12.  ... 58.  65.5 55. ]
 ...
 [41.  55.  58.  ... 45.  75.5 51. ]
 [43.  82.5 65.5 ... 75.5 93.  53. ]
 [70.  68.  55.  ... 51.  53.  94. ]]

+ nowy osobnik 0
+ nowy osobnik 1
+ nowy osobnik 2
+ nowy osobnik 3
+ nowy osobnik 4
+ nowy osobnik 5
+ nowy osobnik 6
+ nowy osobnik 7
+ nowy osobnik 8
+ nowy osobnik 9
Poczatkowa populacja osobnik w:  10
[[64, 9, 31, 50, 74, 70, 77, 26, 59, 68, 13, 10, 79, 51, 14, 73, 80, 58, 41, 72, 53, 3, 56, 75,
    86, 42, 45, 15, 39, 20, 23, 67, 82, 11, 84, 27, 34, 54, 33, 92, 49, 24, 7, 6, 95, 99, 48, 81,
    93, 47, 65, 97, 43, 46, 66, 40, 16, 4, 12, 89, 35, 5, 29, 19, 52, 96, 2, 57, 98, 78, 17, 36,
    83, 76, 25, 38, 55, 0, 63, 44, 30, 90, 37, 88, 69, 91, 28, 61, 22, 62, 71, 87, 60, 21, 32, 85,
     1, 18, 8, 94], [53, 9, 69, 19, 64, 95, 6, 31, 37, 17, 38, 47, 33, 50, 18, 11, 34, 87, 79, 12,
     70, 86, 60, 91, 57, 93, 39, 41, 62, 74, 89, 76, 42, 88, 56, 8, 54, 80, 85, 29, 21, 43, 22,
    72, 97, 77, 14, 7, 78, 3, 45, 44, 15, 65, 28, 2, 32, 4, 99, 48, 27, 59, 5, 0, 73, 40, 84, 55,
    1, 35, 51, 25, 30, 13, 36, 61, 98, 66, 23, 67, 96, 46, 20, 49, 52, 82, 83, 10, 63, 68, 58, 24,
     75, 92, 94, 71, 90, 16, 81, 26], [10, 34, 65, 40, 68, 20, 55, 4, 69, 15, 17, 51, 46, 86, 73,
    41, 60, 3, 82, 79, 32, 89, 63, 57, 64, 43, 45, 6, 75, 8, 72, 44, 9, 54, 61, 42, 24, 47, 12,
    83, 85, 53, 98, 58, 27, 80, 28, 67, 56, 59, 14, 31, 5, 76, 30, 52, 71, 23, 81, 97, 21, 94, 35,
     99, 95, 39, 19, 84, 91, 96, 0, 26, 1, 36, 22, 2, 74, 25, 48, 16, 92, 70, 11, 7, 78, 13, 77,
    38, 90, 33, 37, 49, 88, 50, 62, 18, 93, 87, 29, 66], [26, 70, 98, 39, 45, 2, 71, 53, 54, 51,
    48, 68, 60, 3, 23, 61, 46, 32, 79, 64, 93, 63, 74, 82, 56, 76, 49, 73, 85, 80, 31, 94, 69, 95,
     66, 13, 87, 35, 90, 75, 30, 44, 65, 41, 0, 91, 12, 24, 99, 88, 81, 10, 89, 40, 86, 78, 34,
    43, 59, 4, 28, 25, 50, 7, 42, 5, 6, 9, 62, 36, 83, 29, 20, 16, 1, 19, 8, 38, 14, 92, 58, 15,
    22, 72, 84, 18, 96, 37, 55, 77, 21, 52, 11, 33, 67, 97, 47, 17, 27, 57], [9, 15, 79, 34, 43,
    69, 18, 41, 50, 42, 56, 8, 40, 80, 68, 32, 65, 7, 82, 92, 62, 54, 29, 52, 23, 86, 85, 99, 3,
    89, 58, 6, 95, 55, 30, 22, 35, 44, 93, 28, 49, 51, 59, 97, 72, 36, 87, 73, 46, 21, 19, 66, 26,
     67, 81, 76, 16, 25, 48, 1, 71, 45, 63, 17, 91, 38, 33, 2, 14, 20, 60, 94, 98, 39, 90, 96, 88,
     84, 78, 64, 13, 77, 37, 10, 5, 4, 70, 61, 27, 75, 11, 57, 31, 12, 24, 0, 47, 83, 74, 53],
    [51, 30, 89, 31, 14, 45, 52, 24, 38, 18, 78, 75, 35, 4, 82, 98, 12, 28, 6, 83, 29, 77, 19, 74,
     32, 55, 49, 85, 42, 8, 67, 27, 72, 53, 0, 17, 92, 10, 69, 94, 91, 97, 1, 71, 46, 7, 50, 34,
    56, 39, 21, 33, 26, 81, 36, 41, 95, 64, 37, 22, 43, 99, 16, 88, 2, 40, 25, 60, 73, 13, 79, 68,
     3, 65, 61, 86, 90, 23, 80, 57, 15, 70, 66, 48, 87, 44, 9, 58, 76, 59, 5, 63, 93, 47, 11, 62,
    20, 54, 96, 84], [58, 63, 50, 93, 16, 81, 79, 76, 51, 55, 18, 35, 97, 45, 77, 53, 82, 3, 47,
    36, 44, 48, 83, 42, 41, 73, 7, 70, 33, 61, 60, 46, 24, 59, 31, 88, 43, 80, 92, 10, 69, 5, 72,
    99, 34, 11, 20, 2, 29, 64, 13, 15, 85, 14, 27, 71, 17, 32, 96, 26, 94, 22, 25, 90, 67, 12, 74,
     98, 9, 19, 56, 39, 49, 66, 78, 30, 38, 91, 6, 52, 37, 54, 8, 95, 68, 28, 21, 4, 23, 57, 84,
    86, 89, 65, 75, 62, 0, 87, 40, 1], [49, 70, 21, 87, 1, 67, 95, 98, 50, 66, 24, 3, 89, 58, 39,
    9, 6, 85, 57, 46, 10, 63, 38, 93, 88, 54, 26, 15, 44, 73, 42, 43, 30, 22, 20, 37, 53, 48, 99,
    72, 65, 76, 61, 60, 51, 55, 29, 4, 35, 91, 80, 59, 45, 14, 41, 83, 75, 8, 25, 64, 34, 74, 78,
    0, 56, 16, 40, 33, 47, 31, 13, 32, 11, 68, 28, 79, 18, 19, 97, 52, 90, 94, 23, 12, 5, 2, 84,
    86, 17, 62, 77, 69, 81, 96, 7, 71, 82, 27, 36, 92], [61, 81, 31, 7, 89, 74, 49, 83, 17, 85,
    14, 68, 99, 66, 27, 79, 50, 96, 26, 24, 11, 57, 51, 12, 60, 52, 18, 87, 92, 40, 3, 70, 56, 93,
     76, 30, 38, 53, 44, 16, 9, 80, 15, 35, 8, 41, 94, 73, 71, 62, 29, 47, 91, 37, 25, 58, 45, 2,
    4, 97, 28, 84, 65, 19, 98, 5, 34, 54, 82, 95, 33, 42, 1, 88, 39, 75, 0, 48, 77, 90, 13, 10,
    23, 72, 67, 69, 20, 86, 78, 43, 64, 32, 59, 6, 21, 55, 63, 46, 22, 36], [0, 45, 72, 67, 47,
    41, 36, 4, 28, 69, 97, 49, 68, 34, 21, 55, 17, 23, 43, 65, 48, 99, 25, 63, 26, 30, 46, 20, 71,
     89, 58, 91, 66, 24, 2, 15, 52, 6, 10, 82, 11, 95, 1, 94, 3, 13, 84, 56, 92, 40, 96, 29, 16,
```

```
    9, 98, 60, 85, 42, 5, 59, 79, 44, 93, 39, 80, 90, 87, 14, 50, 86, 12, 78, 57, 83, 35, 61, 31,
    18, 75, 77, 33, 88, 32, 81, 53, 8, 51, 64, 19, 76, 37, 22, 62, 38, 27, 74, 70, 73, 7, 54]]
Ilosc osobnikow w populacji: 10
0
64 -> 9
67.5
1
9 -> 31
61.0
2
31 -> 50
41.0
3
50 -> 74
82.5
.
.
.
Koszt minimalny/najkrotsza trasa 4618.5
Koszt minimalny/najkrotsza trasa dla pokolenia nr 6 : 4618.5
Koszt minimalny/najkrotsza trasa w ostatnim pokoleniu populacji: 4618.5
MBP-Kasia:komiwojazer_kod kasia$
```

Listing 2: Uruchomienie programu dla 10 osobników i 6 pokoleń

Cały output można zapisać do pliku loga.

```
MBP-Kasia:komiwojazer_kod kasia$ python3 komiwojazer.py 10 6 > TSP_10osob_6gen.log
MBP-Kasia:komiwojazer_kod kasia$
```

Listing 3: Output do loga

Listings

Spis rysunków