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,
  - tworzona jest pula rodziców,
  - rodzic może krzyżować się po raz drugi z innym osobnikiem lub rodzic już nie może być drugi raz rodzicem i jest usuwany z populacji potencjalnych rodziców
- z puli rodziców po kolei brane są pary do krzyżowania,
  - następuje krzyżowanie osobników - jednopunktowe, z odwróceniem kolejności genów,
  - wybierany jest losowy punkt krzyżowania,

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

  - 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,

- 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
#
from operator import truediv
import sys
import numpy as np
import random
import matplotlib.pyplot as plt
import datetime

# 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(datetime.datetime.now(), "Poczatkowa populacja osobnikow: ", len(population))
    # print(population)
    return population

```

```python
# 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(datetime.datetime.now(), "Koszt minimalny/najkrotsza trasa", minKoszt)
    return minKoszt

# selekcja turniejowa
# Binary tournament selection (BTS)
# enhancement - zaleznie od parametru passThreshold:
# rodzic juz nie moze byc drugi raz rodzicem, usuwam z populacji potencjalnych rodzicow
def binTounamentSelection(population, costTable, passThreshold):
    # do turnieju 1:1 wylosowac osobnikow
    pair = random.sample(range(0, len(population)), 2)
    # print("para", pair)

    # print(datetime.datetime.now(), "IN --- binTounamentSelection -- liczba w populacji", len(
    population))
    # print("population", population)

    if fnDopasowania(population, costTable, pair[0]) < fnDopasowania(population, costTable, pair
    [1]):
        rodzic = population[pair[0]]
        # print("rodzic to osobnik nr", pair[0], population[pair[0]])
        if passThreshold == False:
            # print("osobnik usuniety nr", pair[1], population[pair[1]])
            del population[pair[1]]     # osobnik usuniety przez fn dopasowania
    else:
        rodzic = population[pair[1]]
        # print("else rodzic to osobnik nr", pair[1], population[pair[1]])
        if passThreshold == False:
            # print("else osobnik usuniety nr", pair[0], population[pair[0]])
            del population[pair[0]]     # osobnik usuniety przez fn dopasowania

    if passThreshold == True:
        # print("======= usun pare")
        if pair[0] < pair[1]:
            del population[pair[1]] # juz nie moze byc drugi raz rodzicem, usuwam z populacji
    potencjalnych rodzicow
            del population[pair[0]] # osobnik usuniety przez fn dopasowania
        else:
            del population[pair[0]] # osobnik usuniety przez fn dopasowania
            del population[pair[1]] # juz nie moze byc drugi raz rodzicem, usuwam z populacji
    potencjalnych rodzicow

    # print(datetime.datetime.now(), "OUT --- binTounamentSelection -- liczba w populacji", len(
    population))
    # print("population", population)
```

```
    # print(datetime.datetime.now(), "OUT --- binTounamentSelection -- liczba rodzicow SO FAR",
    len(rodzic))
    # print("rodzic", rodzic)

    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):
        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, passThreshold):

```

```
    print(datetime.datetime.now(), "----> na wejsciu population, liczba osobnikow:", len(
    population))
    print(datetime.datetime.now(), "----> na wejsciu rodzicow, liczba osobnikow:", len(rodzice))

    # wybranie puli rodzicow
    while len(population) >= 2:
        rodzice.append(binTounamentSelection(population, costTable, passThreshold))

    print(datetime.datetime.now(), "Pula rodzicow, liczba osobnikow:", len(rodzice))
    print(datetime.datetime.now(), "ile osobnikow zostalo w populacji poczatkowej - wygina!:", len
    (population))

    # print(datetime.datetime.now(), "----> PRZED CROSSOVER population, liczba osobnikow:", len(
    population))
    # print(datetime.datetime.now(), "----> PRZED CROSSOVER rodzicow, liczba osobnikow:", len(
    rodzice))

    newGeneration = krzyzowanieRodzicow(rodzice)

    print(datetime.datetime.now(), "DZIECI po krzyzowanie Rodzicow ---> new Generation", len(
    newGeneration))

    # print(datetime.datetime.now(), "----> na OUT population, liczba osobnikow:", len(population)
    )
    # print(datetime.datetime.now(), "----> na OUT rodzicow, liczba osobnikow:", len(rodzice))

    # 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, passThreshold):
    rodzice = []
    newGeneration = []

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

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

    # 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(datetime.datetime.now(), "za duzy mutationRate!")

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

    return newGeneration

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

    print("Liczba miast:", N)
    print("Liczba osobnikow poczatkowych:", ile_osobnikow)
    print("Liczba pokolen:", generations)
    passThreshold = False

    # 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(datetime.datetime.now(), "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(datetime.datetime.now(), "Koszt minimalny/najkrotsza trasa w poczatkowej populacji:",
    minKoszt)

    progress = []
    progress.append(minKoszt)

    print(datetime.datetime.now(), "Tworzenie kolenych pokolen")
    nowePokolenie = population.copy()
    print(datetime.datetime.now(), "generations -- liczba osobnik w w 0 -> 1 pokoleniu", len(
    nowePokolenie))
    # print("nowePokolenie = population", nowePokolenie)

    for i in range(0, generations):
        print(datetime.datetime.now(), "-- pokolenie nr", i+1)
        if threshold > 0 and threshold >= i+1:
            passThreshold = True
        nowePokolenie = nextGeneration(nowePokolenie, costTable, passThreshold)
        # print("generations -----> nowePokolenie", nowePokolenie)
        print(datetime.datetime.now(), "-- generations -- liczba osobnik w w nowym pokoleniu",
    len(nowePokolenie))
        minKoszt = minKosztPodrozy(nowePokolenie, costTable)
        print(datetime.datetime.now(), "Koszt minimalny/najkrotsza trasa dla pokolenia nr", i+1 ,"
    :", minKoszt)
        progress.append(minKoszt)

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

    plt.plot(progress)
    plt.title('Problem komiwojazera')
    plt.ylabel('Koszt podrozy')
    plt.xlabel('Pokolenie')
    plt.savefig('komiwojazer_cost_per_generation.png')
    plt.show()

    return 0

```

```
# ------------------------------ execute genetic algorithm --------------------------------

# ile miast, constant, default=100
# ile osobnikow, default=50
# ile pokolen, default=500

# zaleznie od threshold,
# rodzic juz nie moze byc drugi raz rodzicem, usuwany z populacji potencjalnych rodzicow:
# threshold = 0
# powyzej tej liczby pokolen bedzie usuwany rodzic:
# threshold = 15

# argumenty podane z command line'a
ile_osobnikow = int(sys.argv[1])
generations = int(sys.argv[2])
threshold = int(sys.argv[3])

# run everything and draw a plot: Koszt podrozy per Generation
geneticAlgorithm(ile_osobnikow, generations, threshold)

#
```

Listing 1: Kod źródłowy pliku komiwojazer.py

3 Przykład

3.1 Założenia i wnioski

Dla zadanej liczby miast (ustanowione w kodzie na N=100) oraz osobników początkowych, wygenerowana zostanie zadana liczba pokoleń. Na wyjściu otrzymujemy znalezioną ścieżkę o minimalnym koszcie oraz wyświetlony wykres koszt per pokolenie.

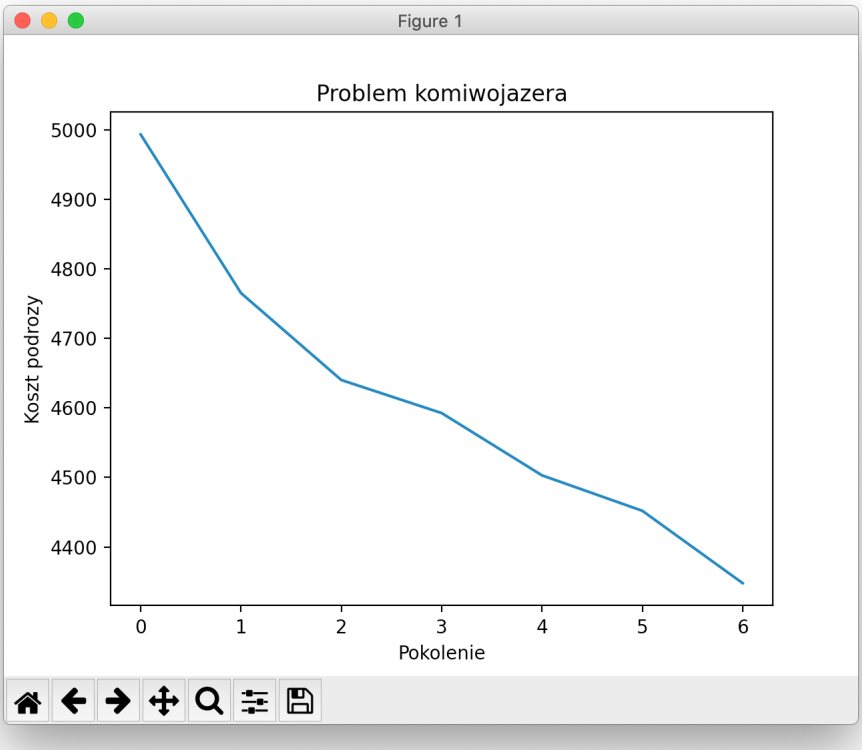
Jeśli chodzi o selekcję turniejową, w niniejszej implementacji, z uwagi na rosnącą liczbę osobników, wprowadzono parametr "threshold" (podawany jako argument uruchomienia programu), aby kontrolować wielkość populacji, zapobiec jej nagłemu wzrostowi lub wymieraniu. Podczas, gdy zaimplementowano pozostawianie w turnieju osobnika-zwycięzcy, a jednocześnie usuwanie osobnika, który "przegrał" selekcję turniejową, wzrost liczby osobników znacząco obniżał wydajność programu. W trakcie selekcji, po przeniesieniu osobnika do następnego pokolenia nie był on usuwany z turnieju, i mógł on dalej uczestniczyć w selekcji turniejowej konkurując z innymi osobnikami. Osobnik, który wygrywał z kilkoma innymi osobnikami mógł rozmnażać się wielokrotnie. Wydaje się ze dzięki temu wyniki są lepsze (więcej lepszych potomków powstało), aniżeli kiedy osobnik był w puli tylko raz. Jednakże algorytm stawał się niewydajny przy liczbie pokoleń > 15 (osobników 50); policzenie kolejnego pokolenia zaczęło zajmować kilkanaście minut i więcej, aż dla pokoleń 17/18 > 30 minut. Przy zaimplementowanej opcji usuwania osobnika-zwycięzcy z turnieju, okazuje się, że wydajność jest nieporównywalnie lepsza, ale populacja stopniowo wymiera. Jednocześnie wyniki są nie tak dobre przy niewielkiej liczbie pokoleń w porównaniu w opcja bez eliminacji wielokrotnego osobnika-zwycięzcy. Wprowadzony parametr "threshold" oznacza, że do podanej liczby pokoleń osobnik-zwycięzca będzie pozostawał w turnieju, a po przekroczeniu tego progu będzie z niego usuwany. Wartość "0" oznacza, że osobnik-zwycięzca pozostaje w turnieju zawsze.

3.2 Przykładowe uruchomienia programu

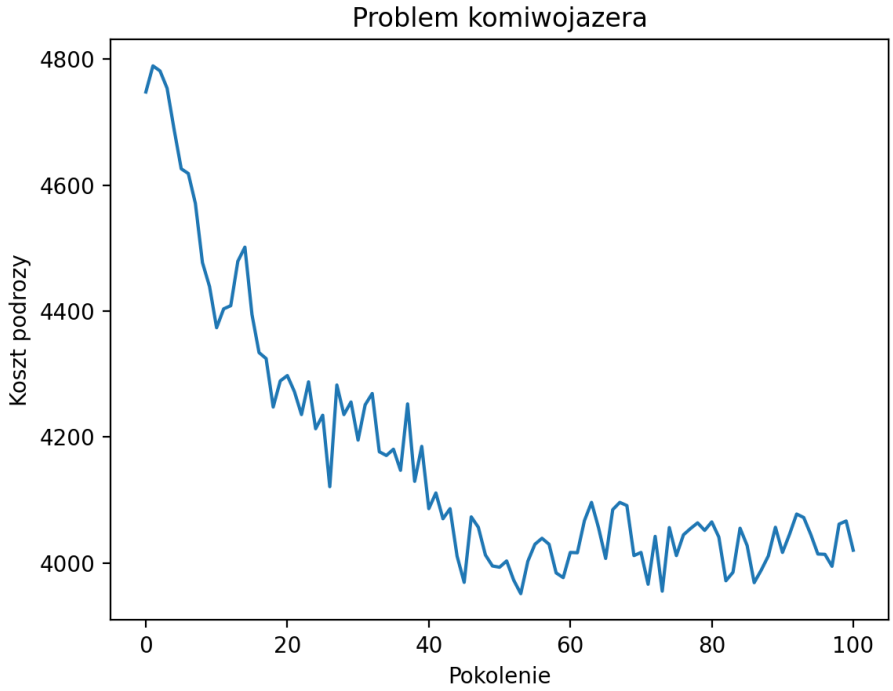
```
MBP-Kasia:komiwojazer_kod kasia$ python3 komiwojazer.py 5 2
Liczba miast: 10
Liczba osobnikow poczatkowych: 5
Liczba pokolen: 2

-------- Tabela kosztow podrozy (10$ - 99$) --------
[[16.  56.  56.  42.5 32.5 62.5 92.  70.  53.  62.5]
 [56.  49.  67.5 24.5 68.  18.5 43.5 65.  29.  39.5]
 [56.  67.5 30.  86.5 36.  51.5 33.5 71.  53.  50. ]
 [42.5 24.5 86.5 84.  53.  69.5 67.5 73.5 93.  37.5]
 [32.5 68.  36.  53.  19.  69.  35.5 54.  61.5 57. ]
 [62.5 18.5 51.5 69.5 69.  69.  43.5 60.  45.5 90. ]
 [92.  43.5 33.5 67.5 35.5 43.5 34.  63.  51.5 41.5]
 [70.  65.  71.  73.5 54.  60.  63.  14.  35.  56. ]
 [53.  29.  53.  93.  61.5 45.5 51.5 35.  30.  54. ]
 [62.5 39.5 50.  37.5 57.  90.  41.5 56.  54.  31. ]]
```

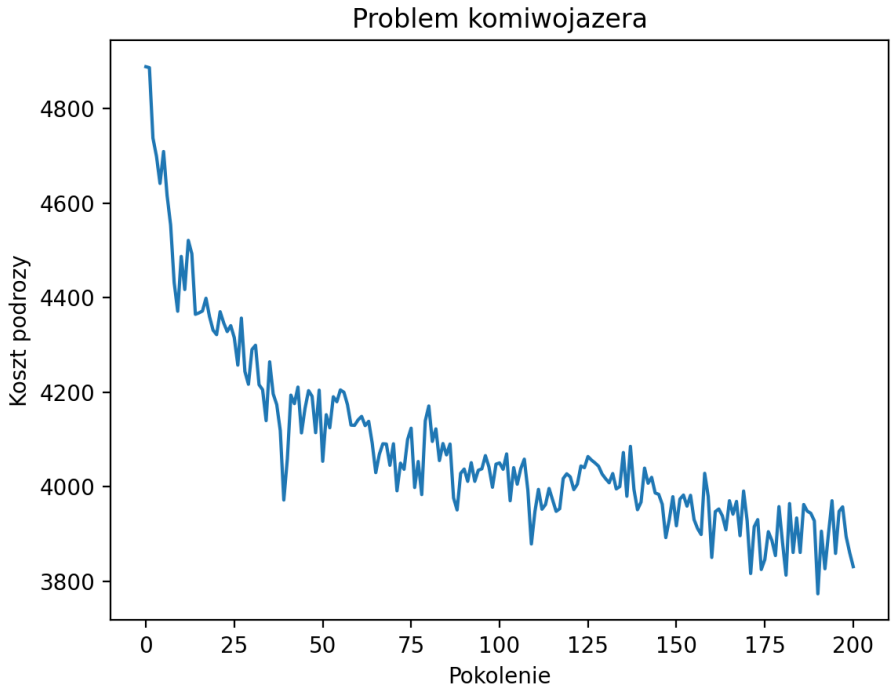
Rysunek 1: Tabela kosztów podróży dla 10 miast. Źródło: własne



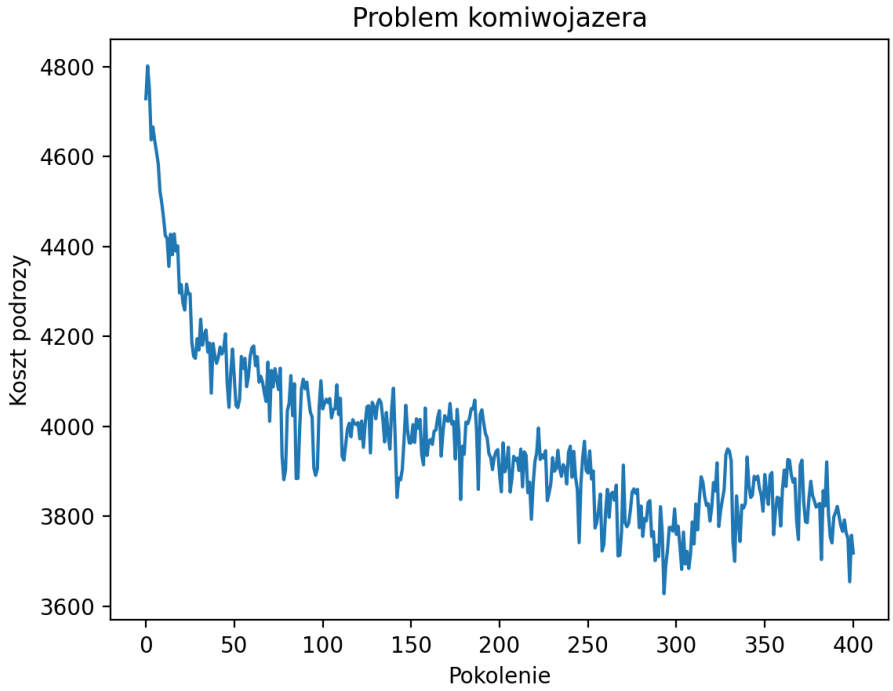
Rysunek 2: Koszt per pokolenie, 10 osobników początkowych, 6 pokoleń. Źródło: własne



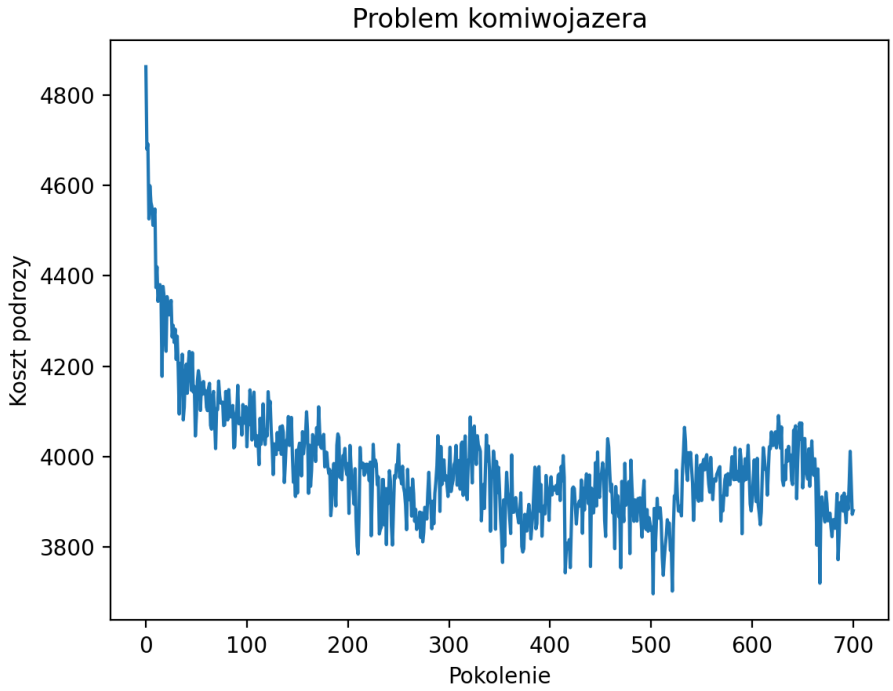
Rysunek 3: Koszt per pokolenie, 300 osobników początkowych, 100 pokoleń, threshold=50. Źródło: własne



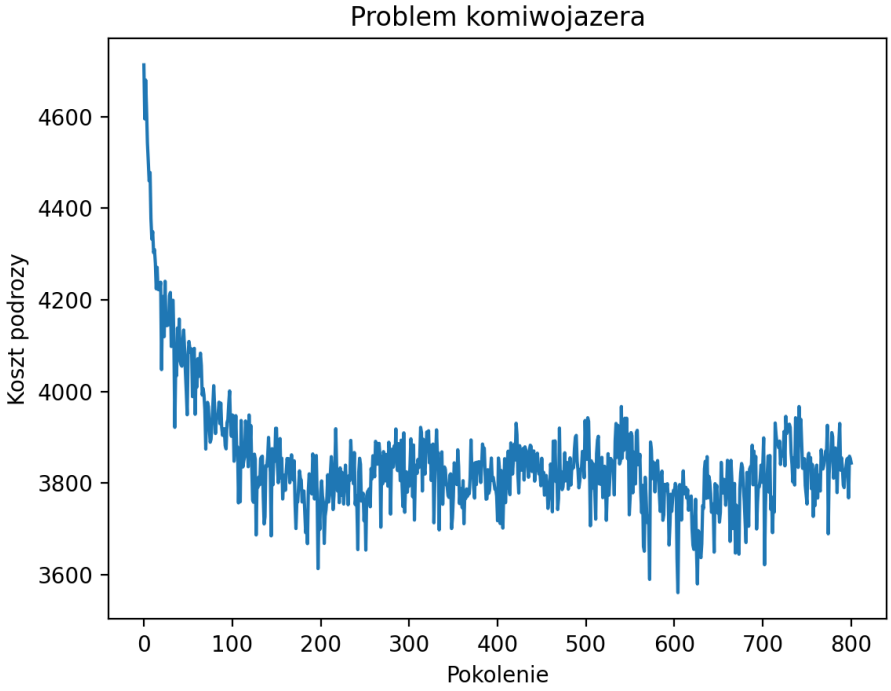
Rysunek 4: Koszt per pokolenie, 500 osobników początkowych, 200 pokoleń, threshold=1. Źródło: własne



Rysunek 5: Koszt per pokolenie, 1000 osobników początkowych, 400 pokoleń, threshold=490. Źródło: własne



Rysunek 6: Koszt per pokolenie, 1500 osobników początkowych, 700 pokoleń, threshold=100. Źródło: własne



Rysunek 7: Koszt per pokolenie, 2000 osobników początkowych, 800 pokoleń, threshold=100. Źródło: własne

3.3 Log z uruchomień

Logi z przykładowych uruchomień znajdują w repozytorium Github:
`https://github.com/kasia-gruszczynska/TSP-Komiwojazer`

Cały output można zapisać do pliku loga.

```
MBP-Kasia:komiwojazer_kod kasia$ python3 komiwojazer.py 50 10 1 > TSP_10osob_6gen.log
MBP-Kasia:komiwojazer_kod kasia$
```

Listing 2: Output do loga

Poniżej zamieszczono przykładowe fragmenty logów, wycinki z konsoli.

```
MBP-Kasia:komiwojazer_kod kasia$ python3 komiwojazer.py 1000 400 490
Liczba miast: 100
Liczba osobnikow poczatkowych: 1000
Liczba pokolen: 400

-------- Tabela kosztow podrozy (10$ - 99$) --------
[[57.  52.  40.5 ... 56.5 76.  69. ]
 [52.  47.  55.5 ... 62.  61.5 58.5]
 [40.5 55.5 55.  ... 65.5 72.5 33. ]
 ...
 [56.5 62.  65.5 ... 52.  35.  52.5]
 [76.  61.5 72.5 ... 35.  63.  47.5]
 [69.  58.5 33.  ... 52.5 47.5 37. ]]

2022-02-28 03:40:32.596536 Poczatkowa populacja osobnikow:  1000
2022-02-28 03:40:32.596595 Ilosc osobnikow w populacji: 1000
2022-02-28 03:40:32.656713 Koszt minimalny/najkrotsza trasa 4728.5
2022-02-28 03:40:32.656843 Koszt minimalny/najkrotsza trasa w poczatkowej populacji: 4728.5
2022-02-28 03:40:32.656856 Tworzenie kolenych pokolen
2022-02-28 03:40:32.656875 generations -- liczba osobnik w w 0 -> 1 pokoleniu 1000
2022-02-28 03:40:32.656889 -- pokolenie nr 1
2022-02-28 03:40:32.656900 ----> na wejsciu population, liczba osobnikow: 1000
2022-02-28 03:40:32.656908 ----> na wejsciu rodzicow, liczba osobnikow: 0
2022-02-28 03:40:32.725801 Pula rodzicow, liczba osobnikow: 500
2022-02-28 03:40:32.725851 ile osobnikow zostalo w populacji poczatkowej - wygina!: 0
2022-02-28 03:40:32.786275 DZIECI po krzyzowanie Rodzicow ---> new Generation 998
2022-02-28 03:40:32.786414 -- generations -- liczba osobnik w w nowym pokoleniu 998
2022-02-28 03:40:32.846909 Koszt minimalny/najkrotsza trasa 4801.5
2022-02-28 03:40:32.847040 Koszt minimalny/najkrotsza trasa dla pokolenia nr 1 : 4801.5
2022-02-28 03:40:32.847057 -- pokolenie nr 2
2022-02-28 03:40:32.847068 ----> na wejsciu population, liczba osobnikow: 998
2022-02-28 03:40:32.847076 ----> na wejsciu rodzicow, liczba osobnikow: 0
2022-02-28 03:40:32.916162 Pula rodzicow, liczba osobnikow: 499
2022-02-28 03:40:32.916209 ile osobnikow zostalo w populacji poczatkowej - wygina!: 0
2022-02-28 03:40:32.990722 DZIECI po krzyzowanie Rodzicow ---> new Generation 996
2022-02-28 03:40:32.991118 -- generations -- liczba osobnik w w nowym pokoleniu 996
2022-02-28 03:40:33.051965 Koszt minimalny/najkrotsza trasa 4748.0
2022-02-28 03:40:33.052092 Koszt minimalny/najkrotsza trasa dla pokolenia nr 2 : 4748.0
2022-02-28 03:40:33.052107 -- pokolenie nr 3
.
.
.
2022-02-28 03:41:17.710699 -- pokolenie nr 399
2022-02-28 03:41:17.710710 ----> na wejsciu population, liczba osobnikow: 204
2022-02-28 03:41:17.710719 ----> na wejsciu rodzicow, liczba osobnikow: 0
2022-02-28 03:41:17.724034 Pula rodzicow, liczba osobnikow: 102
2022-02-28 03:41:17.724081 ile osobnikow zostalo w populacji poczatkowej - wygina!: 0
2022-02-28 03:41:17.735443 DZIECI po krzyzowanie Rodzicow ---> new Generation 202
2022-02-28 03:41:17.735609 -- generations -- liczba osobnik w w nowym pokoleniu 202
2022-02-28 03:41:17.748988 Koszt minimalny/najkrotsza trasa 3757.5
2022-02-28 03:41:17.749063 Koszt minimalny/najkrotsza trasa dla pokolenia nr 399 : 3757.5
2022-02-28 03:41:17.749079 -- pokolenie nr 400
2022-02-28 03:41:17.749091 ----> na wejsciu population, liczba osobnikow: 202
2022-02-28 03:41:17.749099 ----> na wejsciu rodzicow, liczba osobnikow: 0
2022-02-28 03:41:17.761994 Pula rodzicow, liczba osobnikow: 101
2022-02-28 03:41:17.762046 ile osobnikow zostalo w populacji poczatkowej - wygina!: 0
```

```
2022-02-28 03:41:17.772952 DZIECI po krzyzowanie Rodzicow ---> new Generation 200
2022-02-28 03:41:17.773204 -- generations -- liczba osobnik w w nowym pokoleniu 200
2022-02-28 03:41:17.785097 Koszt minimalny/najkrotsza trasa 3718.0
2022-02-28 03:41:17.785207 Koszt minimalny/najkrotsza trasa dla pokolenia nr 400 : 3718.0
2022-02-28 03:41:17.785235 Koszt minimalny/najkrotsza trasa w ostatnim pokoleniu populacji: 3718.0
MBP-Kasia:komiwojazer_kod kasia$
```

Listing 3: Uruchomienie z konsoli

Listings

Spis rysunków