Michał Wilkosz

Traveling Salesman Problem – Report

1. Introduction

Presented solution was developed in Python language with external libraries such as Matplotlib(for drawing plots) and Numpy (for some calculations). In connection with the necessity to contain used libraries, .exe file size exceed the allowable capacity which is possible to send via mail. Therefore, we provide you a link from which you can download the folder with the entire task. Exe file needs to unarchive .zip folder to work. Execution of provided file takes some time(~3min). Thank you in advance for your understanding.

Link:
https://drive.google.com/file/d/1KStaWSL_lUSG7FcL28SyvN0sSUiJIqOT/view?fbclid=IwAR3jpMahdo48q58lVzywQeTNUcX52O38x137GnSnwJ9WdlKdq0r0aEXRezU

The uploaded file is the solution to the first task, using dataset 3 up to the following date of birth – 30.05.1996. Solution to the second task is presented in this document.

2. Source code

```
#! /usr/bin/env python3
import matplotlib.gridspec as gridspec
import matplotlib.pyplot as plt
from numpy.random import choice
from random import choices
import numpy as np
import itertools
import operator
import random
import copy

class Select():
    def __init__(self):
        self.itaretion=0
        self.parent=0
        self.shortest_route=10000
        self.initial_pop_route = None
        self.current_population={}
        self.crossover_population={}
        self.route_values={}
        self.iteration_plot=[]
        self.shortest_route_plot=[]
```

```
    def genetic_algorithm(self, city_data, P, n, pm, T):

        self.current_population = self.initial_population(P,city_data)
        crossover_size = int(((len(self.current_population)*n)/2))
        self.route_values = {}

        while self.itaretion < T:
            self.selection_operator(self.current_population,n)
            for i in range(crossover_size):
                self.offspring_generator(pm)

            self.current_population.update(self.crossover_population)
            self.crossover_population = {}
            for i in self.current_population:
                self.route_values[
                    round((self.total_distance(self.current_population.get(i))
),3)
                    ] = self.current_population.get(i)
            minimum = min(self.route_values.keys())
            if self.itaretion == 0:
                self.initial_pop_route = minimum
            if minimum < self.shortest_route:
                self.shortest_route = minimum
            self.iteration_plot.append(self.itaretion)
            self.shortest_route_plot.append(self.shortest_route)
            self.itaretion += 1

        print("Shortest possible route to considered traveling salesman proble
m is {}".format(self.shortest_route))
        self.draw_plot()

    def draw_plot(self):
        #Shows the results in graphic form
        print("City data before GA: {}\n".format(city_data))
        print("City data after GA: {}\n".format(self.route_values[self.shortes
t_route]))
        X1,Y1 = zip(*city_data)
        X,Y = zip(*self.route_values[self.shortest_route])
        gs = gridspec.GridSpec(3, 2)
        fig = plt.figure()
        fig.subplots_adjust(top=0.95)
        fig.text(0.5, 0.85, "Initial route: {}".format(self.initial_pop_route)
, ha='center', va='center', fontsize='x-large')
        fig.text(0.5, 0.8   , "Shortest route: {}".format(self.shortest_route)
, ha='center', va='center', fontsize='x-large')
        ax1 = fig.add_subplot(gs[1, 0])
        ax1.xaxis.set_label_position('top')

        for i in range(0,len(city_data)):
```

```
            ax1.plot(X1[i:i+2], Y1[i:i+2],'bo')
        plt.xlabel("City Data",axes=ax1)
        ax2 = fig.add_subplot(gs[1, 1])
        ax2.text(X[0],Y[0],"Start")
        ax2.text(X[-1],Y[-1],"End")

        for i in range(0,len(self.route_values[self.shortest_route])):
            ax2.plot(X[i:i+2], Y[i:i+2],'ro-')
        ax2.plot([X[0],X[-1]], [Y[0],Y[-1]], 'b-', label="Return route")
        plt.legend(loc='upper right')
        plt.xlabel("Route of traveling salesman after GA",axes=ax2)
        ax2.xaxis.set_label_position('top')

        ax3 = fig.add_subplot(gs[2, :])
        ax3.plot(self.iteration_plot,self.shortest_route_plot,'k-')
        plt.ylabel("Route distance",axes=ax3)
        plt.xlabel("Algorithm iteration",axes=ax3)
        plt.show()

    def initial_population(self, p,city_data):
        #Create initial population of chromosome in format dict{key[unique_num
ber]:chromosome}
        p_permutations = {}
        for p in range(0,p):
            p_permutations[p] = self.random_route()
        return p_permutations

    def random_route(self):
        #Create random route for selected chromosome
        route = random.sample(city_data, len(city_data))
        return route

    def selection_operator(self, permutation, crossover_size):
        #Selection operator for crossover via roulette wheel
        self.current_population = permutation.copy()
        dictionary_of_probabilities = {}
        for p in permutation:
            dictionary_of_probabilities[p] = self.total_distance(permutation.g
et(p))
            #Format dict{key[unique_number]:route distance}
        population_size = len(self.current_population)
        #Size of parents population
        for parent in list(self.current_population):
        #Iteration over elements of permutation
            if (len(self.crossover_population)/population_size) < crossover_si
ze:
                #Crossover size
                population_distance = sum(list(dictionary_of_probabilities.val
ues()))
```

```
                #Calculates whole route of salesman
                probability = []
                for i in list(dictionary_of_probabilities.values()):
                    #List of probabilities
                    probability.append(i/population_distance)
                invers_prob = self.inverse_probability(probability)
                #Inverts probability, shortest road got highest probability
                self.parent = choice(list(self.current_population.keys()), 1, 
p=invers_prob)
                #Roulette wheel selection
                self.parent = int(self.parent[0])
                self.crossover_population[self.parent] = self.current_populati
on.get(self.parent)
                #Format dict{key[unique_number]:chromosome}
                dictionary_of_probabilities.pop(self.parent,None)
                self.current_population.pop(self.parent,None)

    def total_distance(self, route):
        #Calculates route between all cities
        distance = 0
        x_distance = []
        y_distance = []
        for i in range(0,len(route)-1):
            x_distance.append(abs(route[i][0] - route[i+1][0]))
            y_distance.append(abs(route[i][1] - route[i+1][1]))
            distance += np.sqrt(pow(x_distance[i],2) + pow(y_distance[i],2))
        distance += np.sqrt(pow((route[0][0]-route[len(route)-
1][0]), 2) + pow((route[0][1] - route[len(route)-1][1]), 2))
        return distance

    def inverse_probability(self, probability_list):
        #Invert weights of routes for roulette selection
        weights = [1.0 / w for w in probability_list]
        sum_weights = sum(weights)
        weights = [w / sum_weights for w in weights]
        return weights

    def offspring_generator(self, pm):
        #Generates two offsprings by 2 parents
        parents = {}
        for i in range(2):
            parents[i] = [random.choice(list(self.crossover_population.keys())
)] #parent key : random number,chromosome : parents[0]=[number]
            parents[i].append(self.crossover_population.get(parents[i][0]))
            self.crossover_population.pop(parents[i][0],None)
        offspring_1 = self.cx_operator(parents[0][1],parents[1][1])
        offspring_2 = self.cx_operator(parents[1][1],parents[0][1])
        mutated_offspring_1 = self.mutate(offspring_1, pm)
        mutated_offspring_2 = self.mutate(offspring_2, pm)
```

```
        self.crossover_population[parents[0][0]] = mutated_offspring_1
        self.crossover_population[parents[1][0]] = mutated_offspring_2

    def cx_operator(self, parent1, parent2):
        p1 = list(parent1)
        p2 = list(parent2)
        lenght = len(p1)
        off = self.empty_offspring(lenght)
        new_off = self.cycle_crossover(off, p1, p2)
        return new_off

    def cycle_crossover(self, off, p1, p2):
        #Crossover operator
        current_index = 0
        new_off = list(off)
        new_off[0] = p1[0]
        # print ("Actual new_off: {} current_index: {}".format(new_off, curren
t_index))
        while None in new_off:
            next_allele_2 = p2[current_index]
            if next_allele_2 not in new_off:
                current_index = p1.index(next_allele_2)
                new_off[current_index] = next_allele_2
                # print ("Actual new_off: {} current_index: {}".format(new_off
, current_index))
            else:
                # print ("Actual new_off: {} current_index: {}".format(new_off
, current_index))
                final_off = self.fill_offspring(new_off, p2)
                # print ("Final off: {}".format(final_off))
                return final_off
        return new_off

    def fill_offspring(self, off, p2):
        fill_off = list(off)
        for i in range(len(fill_off)):
            if fill_off[i] is None:
                fill_off[i] = p2[i]
        return fill_off

    def empty_offspring(self, lenght):
        off = [None for i in range(lenght)]
        return off

    def mutate(self, off, pm):
        #Mutation operator
        new_off = list(off)
        idxs = range(len(off))
        n = len(off)
```

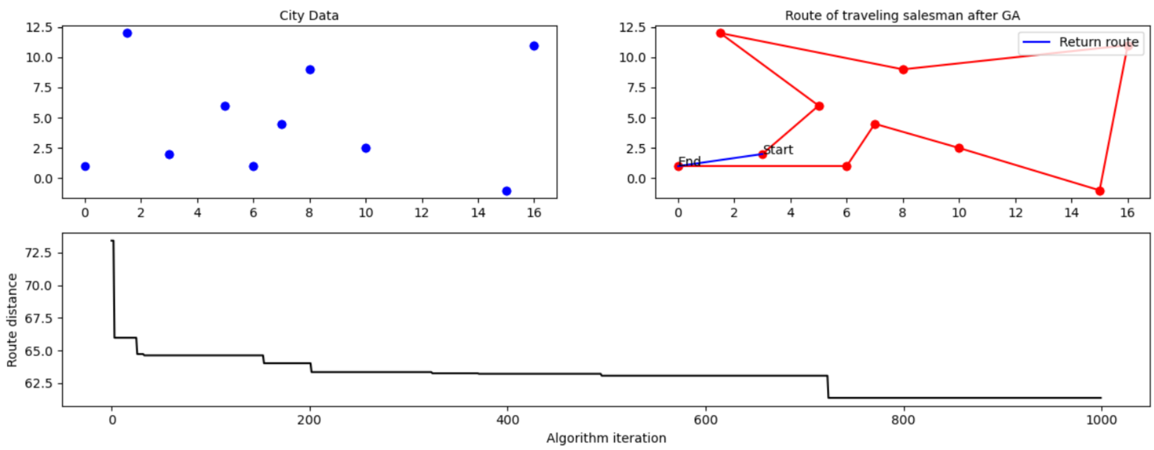
```
        k = int(2*(n*pm))
        indx_list = random.sample(idxs, k)
        mutated_off = self.swap(new_off, indx_list)
        return mutated_off

    def swap(self, off, indx_list):
        swap_tuples = []
        new_off = list(off)
        count = 0
        swaps = []
        for pos in indx_list:
            swaps.append(pos)
            count += 1
            if count == 2:
                swap_tuples.append(swaps)
                count = 0
                swaps = []
        for swp in swap_tuples:
            new_off[swp[0]], new_off[swp[1]] = new_off[swp[1]], new_off[swp[0]
]
        return new_off

def task():
    #Investigation of influence for given parameters
    city_data=[(0,1),(3,2),(6,1),(7,4.5),(15,-1),
               (10,2.5),(16,11),(5,6),(8,9),(1.5,12)]
    #Coordinates of city map by birthdate 30.05.1996r.w

    T = 10
    P_set = [100,300,500]
    n_set = [0.5,0.7,0.9]
    pm_set = [0.1,0.3,0.5]
    results = []

    l_of_sets = [P_set, n_set, pm_set]
    all_possib = list(itertools.product(*l_of_sets))

    for pos in all_possib:
        print ("Calculating for set {}".format(pos))
        select = Select()
        result_list = []
        P = pos[0]
        n = pos[1]
        pm = pos[2]
        shortest = 0
        for i in range(10):
            select.genetic_algorithm(city_data,P,n,pm,T)
            shortest += select.shortest_route
        result_list.append((pos,shortest/10))
```

```
        results.append(result_list)
        print ("Result for given set mean: {}".format(result_list))
    print ("Results: {}".format(results))

if __name__ == "__main__":
    city_data=[(0,1),(3,2),(6,1),(7,4.5),(15,-1),
               (10,2.5),(16,11),(5,6),(8,9),(1.5,12)]
    #Coordinates of city map by birthdate 30.05.1996r.w

    select = Select()

    P = 250
    n = 0.8
    pm = 0.2
    T = 1000

    select.genetic_algorithm(city_data,P,n,pm,T)

    # task()

    pass
```

3. Solution of first task

Initial route: 73.418

Shortest route: 61.376



Commentary:

- First route of traveling salesman was generated randomly.
- The final solution partly depends on the first randomly generated route.
- There are no any crossings during the final route, which may mean that obtained route is the best one at the start from point (3,2).
- The best possible solution was achieved after ~700 iteration.
- With several algorithm runs the final results are sometimes slightly different, however final result is always much smaller than the initial one.

4. Solution of second task

P	***n***	p_{max}	T_{max}	***Results***
100	0.5	0.1	10	69.565
100	0.5	0.3	10	62.181
100	0.5	0.5	10	70.27
100	0.7	0.1	10	66.016
100	0.7	0.3	10	67.226
100	0.7	0.5	10	68.23
100	0.9	0.1	10	67.784
100	0.9	0.3	10	63.018
100	0.9	0.5	10	67.545
300	0.5	0.1	10	68.314
300	0.5	0.3	10	63.220
300	0.5	0.5	10	67.586
300	0.7	0.1	10	66.901
300	0.7	0.3	10	66.582
300	0.7	0.5	10	69.107
300	0.9	0.1	10	65.613
300	0.9	0.3	10	66.694
300	0.9	0.5	10	66.154
500	0.5	0.1	10	63.869
500	0.5	0.3	10	65.603
500	0.5	0.5	10	66.657
500	0.7	0.1	10	67.373
500	0.7	0.3	10	63.869
500	0.7	0.5	10	66.744
500	0.9	0.1	10	61.376
500	0.9	0.3	10	67.663
500	0.9	0.5	10	66.009

Conclusion:

- Population size has the most significant impact on results, as the population increases, the final result usually decreases.
- For P=500, n=0.9, p_{max}=0.1, T_{max}=10 algorithm achieved the best possible result at only 10 iterations.
- The ability to quickly find the minimum route depends on what parents were selected to in selection operator via roulette wheel and then in crossover operator.