IN3050/IN4050 Mandatory Assignment 1: Traveling Salesperson Problem

Rules

Before you begin the exercise, review the rules at this website: https://www.uio.no/english/studies/examinations/compulsory-activities/mn-ifi-mandatory.html (This is an individual assignment. You are not allowed to deliver together or copy/share source-code/answers with others.)

Delivering

Deadline: *Friday, February 19, 2021* **Devilry**: https://devilry.ifi.uio.no

What to deliver?

On the Devilry website, upload one single zipped folder (.zip, .tgz or .tar.gz) which includes:

- PDF report containing:
  - Your name and username (!)
  - Instructions on how to run your program.
  - Answers to all questions from assignment.
  - Brief explanation of what you've done.
  - *Your PDF may be generated by exporting your Jupyter Notebook to PDF, if you have answered all questions in your notebook*
- Source code
  - Source code may be delivered as jupyter notebooks or python files (.py)
- The european cities file so the program will run right away.
- Any files needed for the group teacher to easily run your program on IFI linux machines.

Important: if you weren't able to finish the assignment, use the PDF report to elaborate on what you've tried and what problems you encountered. Students who have made an effort and attempted all parts of the assignment will get a second chance even if they fail initially. This exercise will be graded PASS/FAIL.

Mandatory assingnment 1, IN3050 V2021

Name: Espen Lønes
username: espenlon

I ran all my code inside jupyter notebook. In order to run anny of the code the folowing block must be run first.

In [150…

```python
# imports and reading csv

import numpy as np
import pandas as pd
from itertools import permutations
import time
import matplotlib.pyplot as plt
import random

cities_table = pd.read_csv("european_cities.csv", sep=';')
Cities = cities_table.to_numpy()
```

Introduction

In this exercise, you will attempt to solve an instance of the traveling salesman problem (TSP) using different methods. The goal is to become familiar with evolutionary algorithms and to appreciate their effectiveness on a difficult search problem. You may use whichever programming language you like, but we strongly suggest that you try to use Python, since you will be required to write the second assignment in Python. You must write your program from scratch (but you may use non-EA-related libraries).

	Barcelona	Belgrade	Berlin	Brussels	Bucharest	Budapest
Barcelona	0	1528.13	1497.61	1062.89	1968.42	1498.79
Belgrade	1528.13	0	999.25	1372.59	447.34	316.41
Berlin	1497.61	999.25	0	651.62	1293.40	1293.40
Brussels	1062.89	1372.59	651.62	0	1769.69	1131.52
Bucharest	1968.42	447.34	1293.40	1769.69	0	639.77
Budapest	1498.79	316.41	1293.40	1131.52	639.77	0

Figure 1: First 6 cities from csv file.

Problem

The traveling salesperson, wishing to disturb the residents of the major cities in some region of the world in the shortest time possible, is faced with the problem of finding the shortest tour among the cities. A tour is a path that starts in one city, visits all of the other cities, and then returns to the starting point. The relevant pieces of information, then, are the cities and the distances between them. In this instance of the TSP, a number of European cities are to be visited. Their relative distances are given in the data file, *european_cities.csv*, found in the zip file with the mandatory assignment.

(You will use permutations to represent tours in your programs. If you use Python, the **itertools** module provides a permutations function that returns successive permutations, this is useful for exhaustive search)

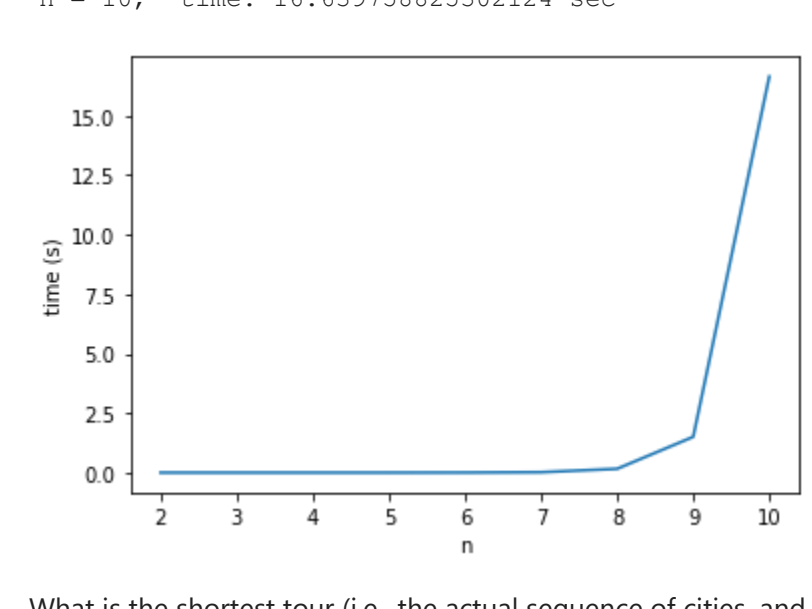
Exhaustive Search

First, try to solve the problem by inspecting every possible tour. Start by writing a program to find the shortest tour among a subset of the cities (say, **6** of them). Measure the amount of time your program takes. Incrementally add more cities and observe how the time increases.

In [300…

```python
def exhaustive(Cities, n):
    """
    Input:
        Cities (numpy array):
            Array of distances between cities.
        n (int):
            Number of cities the subbset.

    Output:
        best_perm (list):
            Shortest path.
        best_distance (float):
            Length of shortest path.
    """
    array = Cities[0:n, 0:n] # cuts down the cities 'table' to the n first cities
    l = list(range(0,n))
    perms = permutations(l) # makes all possible permutations of cities

    best_distance = np.inf # positive float infinity
    best_perm = []
    for perm in perms:
        distance = 0
        for i in range(n):
            distance += array[perm[i-1]][perm[i]]
        if distance < best_distance:
            best_distance = distance
            best_perm = list(perm)

    return best_perm, best_distance;


plot_points = []
r = range(2,11)
for n in r: # change range or make list of values to test different number of cities
    start = time.time()
    print(exhaustive(Cities, n))
    end = time.time()
    print(f"n = {n},  time: {end - start} sec")
    print("")
    plot_points.append(end - start)

plt.plot(r, plot_points)
plt.xlabel("n")
plt.ylabel("time (s)")
plt.show()
```

```
([0, 1], 3056.26)
n = 2,  time: 0.0 sec

([0, 1, 2], 4024.99)
n = 3,  time: 0.0 sec

([0, 1, 2, 3], 4241.89)
n = 4,  time: 0.0 sec

([0, 1, 4, 2, 3], 4983.38)
n = 5,  time: 0.0 sec

([1, 0, 3, 2, 5, 4], 5018.8099999999995)
n = 6,  time: 0.002998828887939453 sec

([2, 6, 3, 0, 1, 4, 5], 5487.889999999999)
n = 7,  time: 0.018003225326538086 sec

([3, 7, 0, 1, 4, 5, 2, 6], 6667.489999999999)
n = 8,  time: 0.16051030158996582 sec

([3, 7, 0, 1, 4, 5, 2, 6, 8], 6678.549999999999)
n = 9,  time: 1.5040991306304932 sec

([8, 3, 7, 0, 1, 9, 4, 5, 2, 6], 7486.309999999999)
n = 10,  time: 16.639758825302124 sec
```



What is the shortest tour (i.e., the actual sequence of cities, and its length) among the first 10 cities (that is, the cities starting with B,C,D,H and I)? How long did your program take to find it? Calculate an approximation of how long it would take to perform exhaustive search on all 24 cities?

In [182…

```python
# Finds and prints shotrest path (must run block with method named exhaustive first)

n = 10
start = time.time()
perm, distance = (exhaustive(Cities, n))
end = time.time()
c = [cities_table.columns[i] for i in range(n)]

print(f"Distace = {distance},\n  Order = {c},\n   Time = {end - start}")
```

```
Distace = 7486.309999999999,
  Order = ['Barcelona', 'Belgrade', 'Berlin', 'Brussels', 'Bucharest', 'Budapest', 'Co
penhagen', 'Dublin', 'Hamburg', 'Istanbul'],
   Time = 16.034034967422485
```

n! permutations for n cities
n = 10 checks 10!=3,628,800 permutations in 16 seconds
Giving 16/3,628,800 = 4.4e-6 seconds per permutation

24! = 620,448,401,733,239,439,360,000
24! * 4.4e-6 = 2,735,663,146,971,955,200 seconds
Whitch is 86,747,309,328 years

Hill Climbing

Then, write a simple hill climber to solve the TSP. How well does the hill climber perform, compared to the result from the exhaustive search for the first **10 cities**? Since you are dealing with a stochastic algorithm, you should run the algorithm several times to measure its performance. Report the length of the tour of the best, worst and mean of 20 runs (with random starting tours), as well as the standard deviation of the runs, both with the **10 first cities**, and with all **24 cities**.

In [313…

```python
def randomPermutation(n):
    """
    Input:
        n (int):
            Number of cities in permutation
    Output:
        permutation (list):
            A random permutation of ints from 0 to n-1
    """
    cities_array = list(range(n))
    permutation = []

    for i in range(n):
        randomCity = cities_array[np.random.randint(len(cities_array))]
        permutation.append(randomCity)
        cities_array.remove(randomCity)

    return permutation

def lengthOfRoute(Cities, permutation):
    """
    Input:
        Cities (numpy array):
            Array of distances between cities.
        permutation (list):
            A random permutation of ints from 0 to n-1

    Outut:
        length (float):
            Length of path described py permutation.
    """
    n = len(permutation)
    nCities = Cities[0:n, 0:n]
    length = 0
    for i in range(len(permutation)):
        length += nCities[permutation[i - 1]][permutation[i]]
    return length

def getNeighbours(permutation):
    """
    Input:
        permutation (list):
            A random permutation of ints from 0 to n-1
    Output:
        neighbours (list):
            List of neighbouring permutations to input permutation.
    """
    neighbours = []
    for i in range(len(permutation)):
        for j in range(i + 1, len(permutation)):
            neighbour = permutation.copy()
            neighbour[i] = permutation[j]
            neighbour[j] = permutation[i]
            neighbours.append(neighbour)
    return neighbours

def bestNeighbour(Cities, neighbours):
    """
    Input:
        Cities (numpy array):
            Array of distances between cities.
        neighbours (list):
            List of neighbouring permutations.

    Output:
        best_neighbour (list):
            Best/shortest path in neighbours set.
        best_length (float):
            Length of  best_neighbour path.
    """
    best_length = lengthOfRoute(Cities, neighbours[0])
    best_neighbour = neighbours[0]
    for neighbour in neighbours:
        length = lengthOfRoute(Cities, neighbour)
        if length < best_length:
            best_length = length
            best_neighbour = neighbour

    return best_neighbour, best_length

def hillClimb(Cities, n):
    """
    Input:
        Cities (numpy array):
            Array of distances between cities.
        n (int):
            Number of cities the subbset.

    Output:
        currentSolution (list):
            Best solution/permutation found by hill climb.
        currentLength (float):
            Length of currentSolution path.
    """
    currentSolution = randomPermutation(n)
    currentLength = lengthOfRoute(Cities, currentSolution)
    neighbours = getNeighbours(currentSolution)

    best_neighbour, best_neighbour_lenght = bestNeighbour(Cities, neighbours)

    while best_neighbour_lenght < currentLength:
        currentSolution = best_neighbour
        currentLength = best_neighbour_lenght
        neighbours = getNeighbours(currentSolution)
        best_neighbour, best_neighbour_lenght = bestNeighbour(Cities, neighbours)

    return currentSolution, currentLength


def task(Cities, n):
    """
    prints best, wors, mean and standard deviation between 20 runs for n cities

    Input:
        Cities (numpy array):
            Array of distances between cities.
        n (int):
            Number of cities we use.
    """
    solutionsWithLength = [None] * 20
    for i in range(20):
        solutionsWithLength[i] = hillClimb(Cities, n)
    sortedSWL = sorted(solutionsWithLength, key=lambda tup: tup[1])

    best = sortedSWL[0][1]
    worst = sortedSWL[-1][1]

    # mean
    s = 0
    for solution in sortedSWL:
        s += solution[1]
    mean = s / len(sortedSWL)

    # standard deviation
    s = 0
    for solution in sortedSWL:
        s += (solution[1] - mean)**2
    s_d = np.sqrt(s / (len(sortedSWL) - 1))

    print(f"n = {n}")
    print("----------")
    print(f" best = {best},\n worst = {worst},\n mean = {mean},\n s_d = {s_d}")

task(Cities, 10)
print("")
task(Cities, 24)
```

```
n = 10
----------
 best = 7486.3099999999995,
 worst = 8391.050000000001,
 mean = 7645.442499999999,
 s_d = 318.394267484679

n = 24
----------
 best = 12870.360000000002,
 worst = 14797.84,
 mean = 13868.622000000003,
 s_d = 539.5886366346706
```

Best from exaustive for n=10 was about: 7486.309999999999
Best from hillclimb for n=10 was about: 7486.309999999999
I ran the 20 * hillclimber many times and got 7486.309999999999 every time

Genetic Algorithm

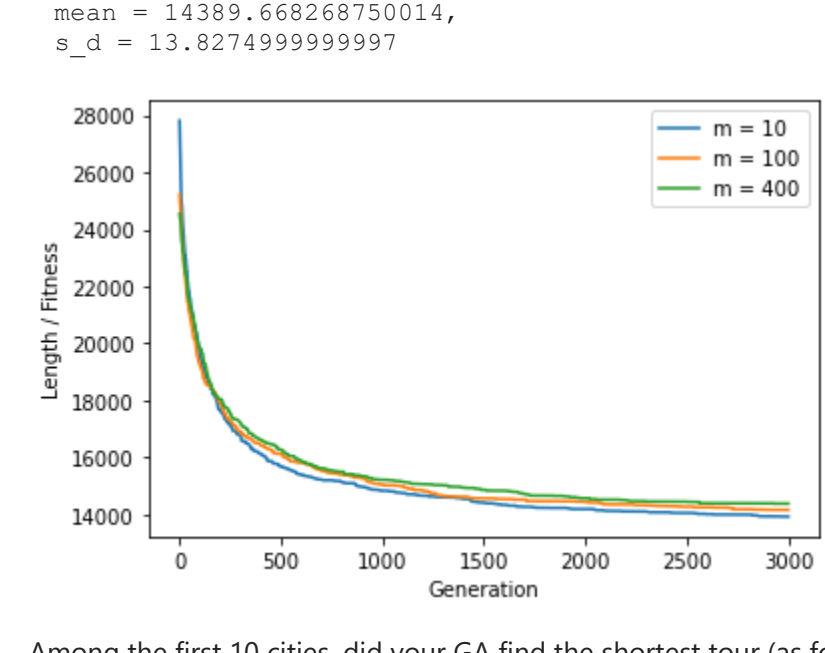
Next, write a genetic algorithm (GA) to solve the problem. Choose mutation and crossover operators that are appropriate for the problem (see chapter 4.5 of the Eiben and Smith textbook). Choose three different values for the population size. Define and tune other parameters yourself and make assumptions as necessary (and report them, of course).

For all three variants: As with the hill climber, report best, worst, mean and standard deviation of tour length out of 20 runs of the algorithm (of the best individual of last generation). Also, find and plot the average fitness of the best fit individual in each generation (average across runs), and include a figure with all three curves in the same plot in the report. Conclude which is best in terms of tour length and number of generations of evolution time.

In [323…

```python
def initialize(m, n):
    """
    Input:
        m (int):
            Size of population.
        n (int):
            Size of a singele sloution.

    Output:
        initialPopulation (list):
            A set of m random soluutions.
    """
    initialPopulation = []

    for i in range(m):
        # Same as randomPermutation()
        cities_array = list(range(n))
        solution = []

        for j in range(n):
            randomCity = cities_array[np.random.randint(len(cities_array))]
            solution.append(randomCity)
            cities_array.remove(randomCity)

        initialPopulation.append(solution)

    return initialPopulation

def fitness(Cities, solution):
    """
    Input:
        Cities (numpy array):
            Array of distances between cities.
        solution (list):
            A random solution (ints from 0 to n-1)

    Outut:
        length (float):
            Length of path described py permutation.
    """
    n = len(solution)
    nCities = Cities[0:n, 0:n]
    length = 0
    for i in range(len(solution)):
        length += nCities[solution[i - 1]][solution[i]]
    return length

def parentSelection(Cities, populationWithFitness, proportion):
    """
    Fitnes-proportionate-selection

    Input:
        populationWithFitness (list):
            The population we select the parents from.
        proportion (float):
            Number between 0.0 and 1.0, determening how large the mating pool is

    Output:
        parents (list):
            The parents used to make next set of offspring.
    """
    l = int(len(population) * proportion) # number of parents chosen (also number of 
    if l % 2 == 0:
        pass
    else:
        l -= 1

    # sort on fitness
    sortedPWF = sorted(populationWithFitness, key=lambda tup: tup[1])

    parents = [None] * l
    for i in range(l):
        parents[i] = sortedPWF.pop(0)[0]
    return parents

def pmx(a, b, start, stop):
    """
    Input:
        a (list):
            The first parent (permutation)
        b (list):
            The second parent (permutation)
        start (int):
            start index for copy from a to child
        stop (int):
            stop index for copy from a to child

    Output:
        child (list):
            New solution formed by pmx crossover of a and b.
    """
    child = [None]*len(a)

    # Copy a slice from first paret:
    child[start:stop] = a[start:stop]

    # Map the same slice in parent b to child using indices from parent a:
    for ind, x in enumerate(b[start:stop]):
        ind += start
        if x not in child:
            while child[ind] != None:
                ind = b.index(a[ind])
            child[ind] = x
    # Copy over the rest from parent b
    for ind, x in enumerate(child):
        if x == None:
            child[ind] = b[ind]

    return child

def pmxPair(a, b):
    """
    Input:
        a (list):
            The first parent (permutation)
        b (list):
            The second parent (permutation)

    Output:
        child_1, child_2 (tuple):
            New solutions formed by pmx crossovers pmx(a, b, start, stop) and pmx(b, 
    """
    half = len(a) // 2
    start = np.random.randint(0, len(a)-half)
    stop = start+half
    return pmx(a, b, start, stop), pmx(b, a, start, stop)


def scrambleMutation(solution):
    """
    Performs a scramble mutation on a permutation solution.
    Pick a subset of genes at random, and randomly rearrange the alleles in those posi

    Input:
        solution (list):
            A solution in a permutation format.

    Output:
        solution (list):
            The solution after the mutation.
    """
    solution_copy = solution.copy()
    locuses = np.random.choice(len(solution), np.random.randint(2, len(solution)), rep
    locuses_list = locuses.tolist()
    for locus in locuses:
        if len(locuses_list) == 1:
            solution[locus] = solution_copy[locuses_list[0]]
        else:
            solution[locus] = solution_copy[locuses_list.pop(np.random.randint(0, len

    return solution

def survivorSelection(solutions, m):
    """
    (mu + lambda)-Selection. We chose the

    Input:
        solutions (list):
            All solutoins we have to chose from
        m (int):
            Size of generation

    Output:

    """
    # sort on fitness
    sort = sorted(solutions, key=lambda tup: tup[1])

    # chose m best solutions
    population = [None] * m
    for i in range(m):
        population[i] = sort.pop(0)

    return population


def runGA(Cities, m, n, g, p_c, p_m, ploting=True):
    """
    Runs the genetic algorithm

    Input:
        Cities (numpy array):
            Array of distances between cities.
        m (int):
            Size of generation
        n (int):
            Size of a singele sloution (also the number of cities we use)
        g (int):
            number of generations
        p_c (float):
            Number between 0.0 and 1.0, determening how large the mating pool is
        p_m (float):
            Number between 0.0 and 1.0, determening how many of the offspring mutate
        ploting (bool):
            True for recording best individual per generation.

    Output:
        populationWithFitness (list):
            The final generation of the genetic algorithm
        bestList (numpy array):
            Fitness of best solution every generation
    """
    # Initialization
    population = initialize(m, n)
    # Set fitnes of population
    populationWithFitness = [None] * len(population)
    for i in range(len(population)):
        populationWithFitness[i] = (population[i], fitness(Cities, population[i])) # 

    if ploting:
        bestList = np.zeros(g+1)
        bestList[0] = sorted(populationWithFitness, key=lambda tup: tup[1])[0][1]

    for gen in range(g):
        random.shuffle(populationWithFitness) # shuffles population list
        # parent selection
        parents = parentSelection(Cities, populationWithFitness, p_c)

        # Crossover
        offspring = [None] * len(parents)
        for i in range(0, len(parents), 2):
            offspring[i], offspring[i+1] = pmxPair(parents[i], parents[i+1])

        # Mutation
        for i in range(int(len(offspring) * p_m)):
            offspring[i] = scrambleMutation(offspring[i])

        # Set fitnes of offspring
        offspringWithFitness = [None] * len(offspring)
        for i in range(len(offspring)):
            offspringWithFitness[i] = (offspring[i], fitness(Cities, offspring[i])) # 

        # Combine curent population(with fitness) and offspring(with fitness)
        combinedWithFitness = populationWithFitness + offspringWithFitness

        # Select members of next generation
        populationWithFitness = survivorSelection(combinedWithFitness, m)

        if ploting == True:
            bestList[gen + 1] = sorted(populationWithFitness, key=lambda tup: tup[1])

    if ploting:
        return populationWithFitness, bestList
    else:
        return populationWithFitness


n = 24
g = 3000
p_c = 0.85
p_m = 0.85
ploting = True

mlist = [10, 100, 400]

for m in mlist:
    finalGens = [None] * 20
    bestLists = [None] * 20
    for i in range(20):
        finalGen, bestLists[i] = runGA(Cities, m, n, g, p_c, p_m, ploting)
        finalGens[i] = sorted(finalGen, key=lambda tup: tup[1])

    allBest = [None] * 20
    allWorst = [None] * 20
    for i in range(len(finalGens)):
        allBest[i] = finalGens[i][0][1]
        allWorst[i] = finalGens[i][-1][1]

    allWorst.sort()
    allBest.sort()
    best = allBest[0]
    worst = allWorst[-1]

    # mean
    means = [None] * 20
    for i in range(len(finalGens)):
        s = 0
        for solution in finalGens[i]:
            s += solution[1]
        means[i] = s / len(finalGens[i])

    s = 0
    for mean in means:
        s += mean
    mean = s / len(means)

    # standard deviation
    s_dList = [None] * 20
    for i in range(len(finalGens)):
        s = 0
        for solution in finalGens[i]:
            s += (solution[1] - means[i])**2
        s_dList[i] = np.sqrt(s / (len(finalGens[i]) - 1))


    print(f"m = {m}")
    print("----------")
    print(f" best = {best},\n worst = {worst},\n mean = {mean},\n s_d = {s_d}")
    print("")
    if ploting:
        avgBestList = [0] * len(bestLists[0])
        for bestList in bestLists:
            for i in range(len(bestList)):
                avgBestList[i] += bestList[i]

        for i in range(len(avgBestList)):
            avgBestList[i] = avgBestList[i] / len(bestLists)

        plt.plot(avgBestList, label=f"m = {m}")

if ploting:
    plt.xlabel("Generation")
    plt.ylabel("Length / Fitness")
    plt.legend()
    plt.show()
```

```
m = 10
----------
 best = 12783.720000000001,
 worst = 16031.94,
 mean = 13937.85365,
 s_d = 13.8274999999997

m = 100
----------
 best = 12635.48,
 worst = 15272.830000000002,
 mean = 14162.848020000005,
 s_d = 13.8274999999997

m = 400
----------
 best = 12719.079999999998,
 worst = 15906.6,
 mean = 14389.668268750014,
 s_d = 13.8274999999997
```



Among the first 10 cities, did your GA find the shortest tour (as found by the exhaustive search)? Did it come close?

For both 10 and 24 cities: How did the running time of your GA compare to that of the exhaustive search?

How many tours were inspected by your GA as compared to by the exhaustive search?

In [324…

```python
m = 400
n = 10
g = 3000
p_c = 0.85
p_m = 0.85
ploting = False

finalGens = [None] * 20
bestLists = [None] * 20
for i in range(20):
    finalGen = runGA(Cities, m, n, g, p_c, p_m, ploting)
    finalGens[i] = sorted(finalGen, key=lambda tup: tup[1])

allBest = [None] * 20
for i in range(len(finalGens)):
    allBest[i] = finalGens[i][0][1]
allBest.sort()
best = allBest[0]
print(best)

start_10 = time.time()
runGA(Cities, m, 10, g, p_c, p_m, ploting)
end_10 = time.time()

start_24 = time.time()
runGA(Cities, m, 24, g, p_c, p_m, ploting)
end_24 = time.time()

print(f"Time for 10 cities: {end_10 - start_10}")
print(f"Time for 24 cities: {end_24 - start_24}")
```

```
7486.309999999999
Time for 10 cities: 1.8769638538360596
Time for 24 cities: 2.4740402698516846
```

Distace for first 10 cities as found by exaustive = 7486.309999999999
Distace for first 10 cities as found by GA = 7486.3099999999995
So yes it found the best one.

Time for 10 cities: 1.8769638538360596 seconds
Time for 24 cities: 2.4740402698516846 seconds
For 24 cities it is an estimated speed increase on the scale of 1*10^18

Did not have time to find out how many tours where checked.

Hybrid Algorithm (IN4050 only)

Lamarckian

Lamarck, 1809: Traits acquired in parents' lifetimes can be inherited by offspring. In general the algorithms are referred to as Lamarckian if the result of the local search stage replaces the individual in the population.

Baldwinian

Baldwin effect suggests a mechanism whereby evolutionary progress can be guided towards favourable adaptation without the changes in individual's fitness arising from learning or development being reflected in changed genetic characteristics. In general the algorithms are referred to as Baldwinian if the original member is kept, but has as its fitness the value belonging to the outcome of the local search process.

(See chapter 10 and 10.2.1 from Eiben and Smith textbook for more details. It will also be lectured in Lecure 4)

Task

Implement a hybrid algorithm to solve the TSP: Couple your GA and hill climber by running the hill climber a number of iterations on each individual in the population as part of the evaluation. Test both Lamarckian and Baldwinian learning models and report the results of both variants in the same way as with the pure GA (min, max, mean and standard deviation of the end result and an averaged generational plot). How do the results compare to that of the pure GA, considering the number of evaluations done?

In [1]:

```python
# Implement algorithm here
```