

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-fi-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.fi.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)
 - Any of the python files 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 assignment 1, IN3050 V2021

Name: Espen Lønes

username: espenlon

I chose to write and run all my code inside jupyter notebook.

In order to run any of the code the following block must be run first.

Some of the blocks also require previous blocks in that part of the assignment to be run first.

```
In [1]:  
# 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.

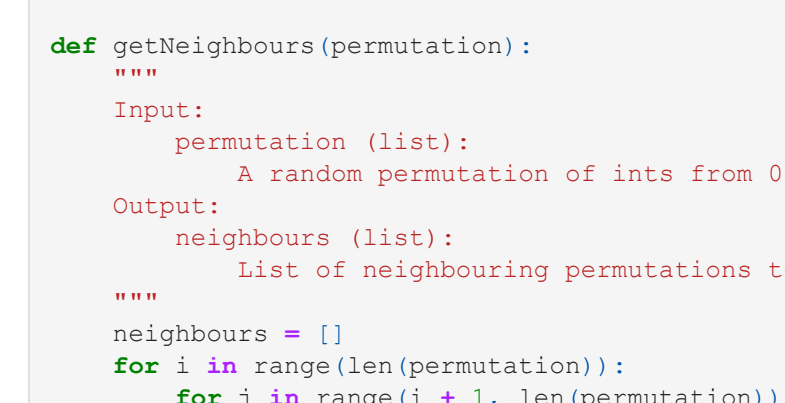
(You will use permutations to represent tours in your programs. If you use Python, the `itertools` module provides a permutations function that returns successive permutations, this is useful for exhaustive search)

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 [2]:  
def exhaustive(Cities, n):  
    """  
    Input:  
        Cities (numpy array):  
            Array of distances between cities.  
        n (int):  
            Number of cities the subset.  
    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-1):  
            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), 5018.8099999999999)  
n = 6, time: 0.001991397857666016 sec  
  
((2, 6, 3, 0, 1, 4, 5), 5467.8899999999999)  
n = 7, time: 0.018016815185546875 sec  
  
((3, 7, 0, 1, 4, 5, 2, 6), 6667.4899999999999)  
n = 8, time: 0.1629819869951172 sec  
  
((3, 7, 0, 1, 4, 5, 2, 6, 8), 6678.5499999999999)  
n = 9, time: 1.498038053325732 sec  
  
((8, 3, 7, 0, 1, 9, 4, 5, 2, 6), 7486.3099999999999)  
n = 10, time: 17.290999174118042 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 would your program take to find it? Calculate an approximation of how long it would take to perform exhaustive search on all 24 cities?

```
In [16]:  
# Finds and prints shortest path (must run block with method named exhaustive first)  
  
n = 10  
perm, distance = exhaustive(Cities, n)  
end = time.time()  
c = [cities_table.columns[i] for i in range(n)]  
  
print(f"Distance = {distance},\n Order = {c},\n Time = (end - start)")  
  
Distance = 7486.3099999999999  
Order = ['Barcelona', 'Belgrade', 'Berlin', 'Brussels', 'Bucharest', 'Budapest', 'Copenhagen', 'Dublin', 'Hamburg', 'Istanbul']  
Time = 17.290999294226685  
  
n! permutations for n cities  
n = 10 checks 10!=3,628,800 permutations in 17.2 seconds  
Giving 17.2/3,628,800 = 4.7e-6 seconds per permutation  
  
24! = 620,448,401,733,239,439,360,000  
24! * 4.7e-6 = 2,940,837,882,994,851,840 seconds  
Which is 93,253,357,527 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 [17]:  
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  
    Output:  
        length (float):  
            Length of path described by permutation.  
    """  
    n = len(permutation)  
    nCities = Cities[0:n, 0:n]  
    length = 0  
    for i in range(len(permutation)-1):  
        length += nCities[permutation[i] - 1][permutation[i+1]]  
    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)-1):  
        for j in range(i + 1, len(permutation)): # swap  
            neighbour = permutation.copy()  
            neighbour[i], neighbour[j] = permutation[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 subset.  
    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_length = bestNeighbour(Cities, neighbours)  
  
    while best_neighbour_length < currentLength:  
        currentSolution = best_neighbour  
        currentLength = best_neighbour_length  
        neighbours = getNeighbours(currentSolution)  
        best_neighbour, best_neighbour_length = bestNeighbour(Cities, neighbours)  
  
    return currentSolution, currentLength  
  
def task(Cities, n):  
    """  
    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"-----")  
    print(f"n = {n}")  
    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 = 7486.3099999999999  
worst = 7737.9500000000001  
mean = 7514.832,  
s_d = 76.60661848554284  
  
n = 24  
-----  
best = 12742.43,  
worst = 16876.93,  
mean = 14504.284,  
s_d = 1104.6454195537376
```

Best from exhaustive for n=10 was about: 7486.3099999999999

Best from hillclimb for n=10 was about: 7486.3099999999999

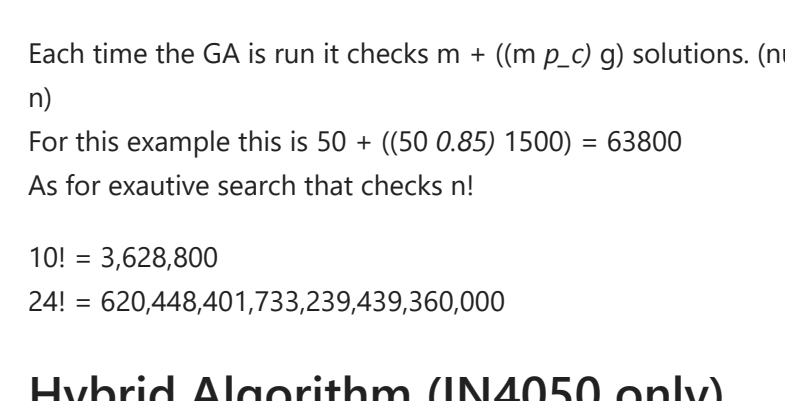
I ran the 20 * hillclimber many times and got 7486.3099999999999 every time

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 [14]:  
def initialize(m, n):  
    """  
    Input:  
        m (int):  
            Size of population.  
        n (int):  
            Size of a single solution.  
    Output:  
        initialPopulation (list):  
            A set of m random solutions.  
    """  
    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)  
    Output:  
        length (float):  
            Length of path described by permutation.  
    """  
    n = len(solution)  
    nCities = Cities[0:n, 0:n]  
    length = 0  
    for i in range(len(solution)-1):  
        length += nCities[solution[i] - 1][solution[i+1]]  
    return length  
  
def parentSelection(Cities, populationWithFitness, proportion):  
    """  
    Input:  
        populationWithFitness (list):  
            The population we select the parents from.  
        proportion (float):  
            Number between 0.0 and 1.0, determining how large the mating pool is  
    Output:  
        parents (list):  
            The parents used to make next set of offspring.  
    """  
    l = int(len(populationWithFitness) * proportion) # number of parents chosen (also  
    if l % 2 != 0:  
        l += 1  
    else:  
        l = 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 parent:  
    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]):  
        if x not in child:  
            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, a, start, stop)  
    """  
    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 positions.  
    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)), replace=True)  
    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(locuses_list)))]  
  
    return solution  
  
def survivorSelection(solutions, m):  
    """  
    (m+lambda)-Selection. We chose the m best solutions and lambda new ones.  
    Input:  
        solutions (list):  
            solutions we have to choose from  
        m (int):  
            Size of generation  
    Output:  
        population (list):  
            The population after selection.  
    """  
    # 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, plotting = True):  
    """  
    Runs the genetic algorithm  
    Input:  
        Cities (numpy array):  
            Array of distances between cities.  
        m (int):  
            Size of generation  
        n (int):  
            Size of a single solution (also the number of cities we use)  
        g (int):  
            number of generations  
        p_c (float):  
            Number between 0.0 and 1.0, determining how large the mating pool is  
        p_m (float):  
            Number between 0.0 and 1.0, determining how many of the offspring mutate  
        plotting (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 fitness of population  
    populationWithFitness = [None] * len(population)  
    for i in range(len(population)):  
        populationWithFitness[i] = (population[i], fitness(Cities, population[i]))  
  
    if plotting:  
        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 fitness of offspring  
        offspringWithFitness = [None] * len(offspring)  
        for i in range(len(offspring)):  
            offspringWithFitness[i] = (offspring[i], fitness(Cities, offspring[i]))  
  
        # Combine current population (with fitness) and offspring (with fitness)  
        combinedWithFitness = populationWithFitness + offspringWithFitness  
  
        # Select members of next generation  
        populationWithFitness = survivorSelection(combinedWithFitness, m)  
  
    if plotting == True:  
        bestList[gen + 1] = sorted(populationWithFitness, key=lambda tup: tup[1])[0][1]  
  
    if plotting:  
        return populationWithFitness, bestList  
    else:  
        return populationWithFitness  
  
n = 10  
g = 1500  
p_c = 0.85  
p_m = 0.85  
plotting = True  
mList = [10, 25, 50]  
  
for m in mList:  
    finalGens = [None] * 20  
    bestLists = [None] * 20  
    for i in range(20):  
        finalGens[i], bestLists[i] = runGA(Cities, m, n, g, p_c, p_m, plotting)  
        finalGens[i] = sorted(finalGens, 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)):  
        for solution in finalGens[i]:  
            s += (solution[1] - mean)**2  
        s_dList[i] = np.sqrt(s / (len(finalGens[i]) - 1))  
  
    s = 0  
    for s_d in s_dList:  
        s += s_d  
    s_d = s / len(s_dList)  
  
    print(f"n = {n}")  
    print(f"-----")  
    print(f"best = {best},\n worst = {worst},\n mean = {mean},\n s_d = {s_d}")  
    print(f"")  
  
    if plotting:  
        avgBestList = [0] * len(bestLists)  
        for i in range(len(bestLists)):  
            avgBestList[i] = bestList[i]  
  
        for i in range(len(avgBestList)):  
            avgBestList[i] = avgBestList[i] / len(bestLists)  
  
        plt.plot(avgBestList, label=f"m = {m}")  
  
    if plotting:  
        plt.xlabel("Generation")  
        plt.ylabel("Length / fitness")  
        plt.legend()  
        plt.show()  
  
# This takes some time 2-3 min  
  
m = 10  
-----  
best = 13225.3100000000003,  
worst = 16601.0,  
mean = 14901.330550000002,  
s_d = 31.23493312275258  
  
m = 25  
-----  
best = 12678.460000000003,  
worst = 15034.69,  
mean = 13873.820039999997,  
s_d = 4.1930919780046574  
  
m = 50  
-----  
best = 12287.07,  
worst = 15338.180000000002,  
mean = 13457.948929999999,  
s_d = 5.02403273581444
```



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 [15]:  
m = 50  
n = 10  
g = 1500  
p_c = 0.85  
p_m = 0.85  
plotting = False  
  
finalGens = [None] * 20  
bestLists = [None] * 20  
for i in range(20):  
    finalGens[i], bestLists[i] = runGA(Cities, m, n, g, p_c, p_m, plotting)  
    finalGens[i] = sorted(finalGens, 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]  
print(best)  
  
start_10 = time.time()  
runGA(Cities, m, 10, g, p_c, p_m, plotting)  
end_10 = time.time()  
  
start_24 = time.time()  
runGA(Cities, m, 24, g, p_c, p_m, plotting)  
end_24 = time.time()  
  
print(f"Time for 10 cities: (end_10 - start_10)")  
print(f"Time for 24 cities: (end_24 - start_24)")  
  
7486.3099999999999  
Time for 10 cities: 2.222034454345703  
Time for 24 cities: 3.7019970417022705  
Distance for first 10 cities as found by exhaustive = 7486.3099999999999  
Distance for first 10 cities as found by GA = 7486.3099999999999  
So yes it found the best one.  
  
Time for 10 cities: 1.8769638538360596 seconds ( 20 if we run it and take best of 20 runs)  
Time for 24 cities: 2.474042698516846 seconds ( 20 if we run it and take best of 20 runs)  
For 24 cities it is an estimated speed increase on the scale of 1*10-18
```

Each time the GA is run it checks $m + (m \cdot p_c \cdot g)$ solutions. (number of solutions checked is independent of n)

For this example this is $50 + (50 \cdot 0.85) \cdot 1500 = 63800$

As for exhaustive search that checks $n!$

$10! = 3,628,800$

$24! = 620,448,401,733,239,439,360,000$

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 methods 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 [11]:  
# I am an IN3050 student :)
```