IN3050/IN4050 Mandatory Assignment 1 - Annica - @annicas : Traveling Salesman 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.)

Especially, notice that you are **not allowed to use code or parts of code written by others** in your submission. We do check your code against online repositories, so please be sure to **write all the code yourself**. Read also the "Routines for handling suspicion of cheating and attempted cheating at the University of Oslo": https://www.uio.no/english/studies/examinations/cheating/index.html By submitting this assignment, you confirm that you are familiar with the rules and the consequences of breaking them.

Delivery

Deadline: Friday, February 24 2023, 23:59

Your submission should be delivered in Devilry. You may redeliver in Devilry before the deadline, but include all files in the last delivery, as only the last delivery will be read. You are recommended to upload preliminary versions hours (or days) before the final deadline.

What to deliver?

You are recommended to solve the exercise in a Jupyter notebook, but you might solve it in a Python program if you prefer.

If you choose Jupyter, you should deliver the notebook. You should answer all questions and explain what you are doing in Markdown. Still, the code should be properly commented. The notebook should contain results of your runs. In addition, you should make a pdf of your solution which shows the results of the runs.

If you prefer not to use notebooks, you should deliver the code, your run results, and a pdf-report where you answer all the questions and explain your work.

Your report/notebook should contain your name and username.

Deliver one single zipped folder (.zip, .tgz or .tar.gz) which contains your complete solution.

Important: if you weren't able to finish the assignment, use the PDF report/Markdown 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.

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 salesman, 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)

Helper code for visualizing solutions

Here follows some helper code that you can use to visualize the plans you generate. These visualizations can **help you check if you are making sensible tours or not**. The optimization algoritms below should hopefully find relatively nice looking tours, but perhaps with a few visible inefficiencies.

In [ ]:
```
%matplotlib inline
import numpy as np
```

```
import matplotlib.pyplot as plt
#Map of Europe
europe_map =plt.imread('map.png')

#Lists of city coordinates
city_coords={"Barcelona":[2.154007, 41.390205], "Belgrade": [20.46,44.79], "Berlin
```

In [ ]:
```
#Helper code for plotting plans
#First, visualizing the cities.
import csv
with open("european_cities.csv", "r") as f:
    data = list(csv.reader(f, delimiter=';'))
    cities = data[0]

fig, ax = plt.subplots(figsize=(10,10))

ax.imshow(europe_map, extent=[-14.56,38.43, 37.697 +0.3 , 64.344 +2.0], aspect = "



# Map (long, lat) to (x, y) for plotting
for city,location in city_coords.items():
    x, y = (location[0], location[1])
    plt.plot(x, y, 'ok', markersize=5)
    plt.text(x, y, city, fontsize=12);
```

In [ ]:
```
#A method you can use to plot your plan on the map.
def plot_plan(city_order):
    fig, ax = plt.subplots(figsize=(10,10))
```

```
    ax.imshow(europe_map, extent=[-14.56,38.43, 37.697 +0.3 , 64.344 +2.0], aspect

    # Map (long, lat) to (x, y) for plotting
    for index in range(len(city_order) -1):
        current_city_coords = city_coords[city_order[index]]
        next_city_coords = city_coords[city_order[index+1]]
        x, y = current_city_coords[0], current_city_coords[1]
        #Plotting a line to the next city
        next_x, next_y = next_city_coords[0], next_city_coords[1]
        plt.plot([x,next_x], [y,next_y])

        plt.plot(x, y, 'ok', markersize=5)
        plt.text(x, y, index, fontsize=12);
    #Finally, plotting from last to first city
    first_city_coords = city_coords[city_order[0]]
    first_x, first_y = first_city_coords[0], first_city_coords[1]
    plt.plot([next_x,first_x],[next_y,first_y])
    #Plotting a marker and index for the final city
    plt.plot(next_x, next_y, 'ok', markersize=5)
    plt.text(next_x, next_y, index+1, fontsize=12);
    plt.show();
```

In [ ]:

```
#Example usage of the plotting-method.
plan = list(city_coords.keys()) # Gives us the cities in alphabetic order
print(plan)
plot_plan(plan)
```

```
['Barcelona', 'Belgrade', 'Berlin', 'Brussels', 'Bucharest', 'Budapest', 'Copenhag
en', 'Dublin', 'Hamburg', 'Istanbul', 'Kiev', 'London', 'Madrid', 'Milan', 'Mosco
w', 'Munich', 'Paris', 'Prague', 'Rome', 'Saint Petersburg', 'Sofia', 'Stockholm',
'Vienna', 'Warsaw']
```

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. Plot the shortest tours you found using the plot_plan method above, for 6 and 10 cities.

In [ ]:

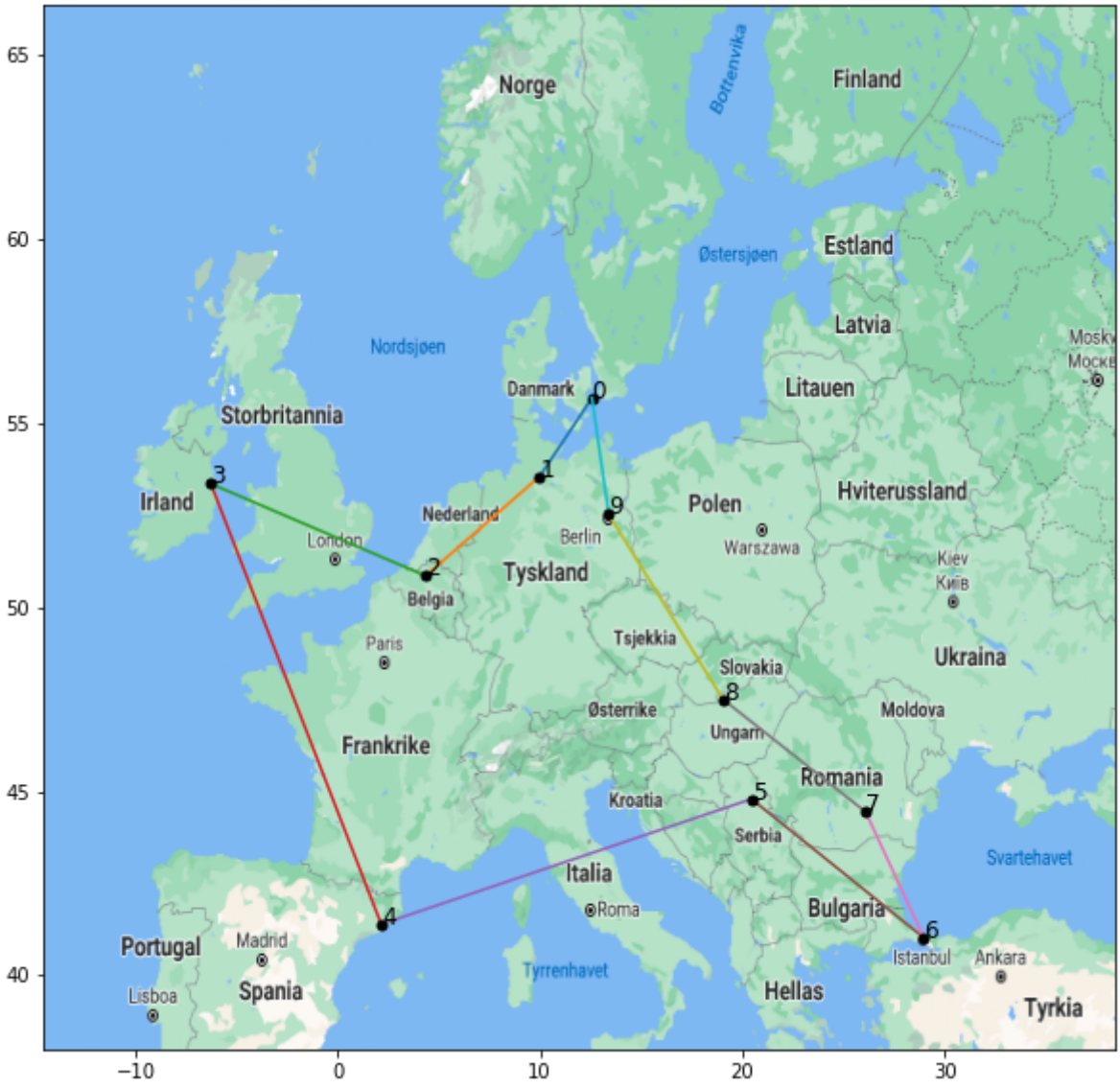
```
# Implement the algorithm here
import itertools

def exhaustiveSearch(n):
    # Create a dictionary of distances between cities
    distances = {}
    for i in range(n):
        for j in range (n):
            # Only calculate distance between different cities
            if cities[i] != cities[j]:
                distances[(cities[i], cities[j])] = data[i+1][j]

    # Get a list of all permutations of n cities
    nCities = cities[:n]
    combs = list(itertools.permutations(nCities))
```

```
    # Set initial values for shortest distance and shortest path
    shortest = float("inf")
    shortestPath = None

    # Loop through each possible path and calculate the total distance    shortest
    shortestPath = None
    for path in combs:
        dist = 0
        for i in range(n-1):
            city = path[i]
            city2 = path[i+1]
            dist += float(distances[(city, city2)])
            if dist > shortest:
                # Stop calculating the distance if it's already longer than the cu
                break
        # Add the distance from the last city to the first city to complete the pa
        dist += float(distances[(city2, path[0])])
        # Update the shortest path and distance if this is shorter
        if dist < shortest:
            shortest = dist
            shortestPath = path
    return shortestPath, shortest
```

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 [ ]:
```
# Answer
import time

start = time.time()
path, distance = exhaustiveSearch(10)
end = time.time()
ES_time = end - start
print("Executed time for Exhaustive search: ",  "{:.4f}".format(ES_time))

print("The shortest tour found : ", path, "with distance ", "{:.4f}".format(distan
plot_plan(path)
```

```
Executed time for Exhaustive search:  16.9549
The shortest tour found :  ('Copenhagen', 'Hamburg', 'Brussels', 'Dublin', 'Barcel
ona', 'Belgrade', 'Istanbul', 'Bucharest', 'Budapest', 'Berlin') with distance  74
86.3100
```



Hill Climbing

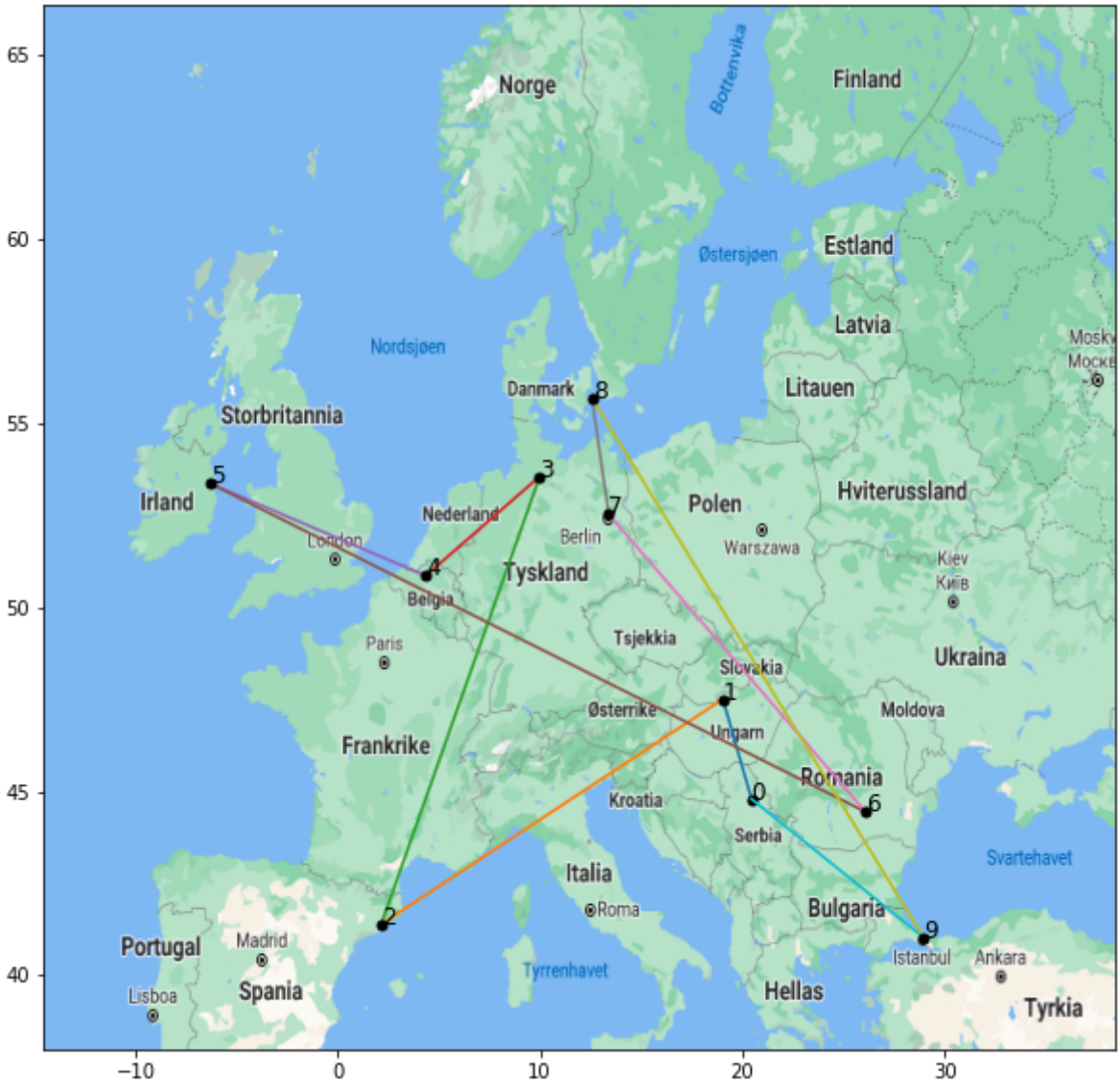
Then, write a simple hill climber to solve the TSP. How well does the hill climber perform, compared to the result from the exhaustive search for the first **10 cities**? Since you are dealing with a stochastic algorithm, you should run the algorithm several times to measure its performance. Report the 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**. Plot one of the the plans from the 20 runs for both 10 cities and 24 cities (you can use plot_plan).

In [ ]:
```python
# Implement the algorithm here
import random
# Collecting distances in dict
distances = {}
for i in range(24):
    for j in range (24):
        if cities[i] != cities[j]:
            distances[(cities[i], cities[j])] = data[i+1][j]

# Method for calculating dist (also used in part3)
def get_distance(path, n=24, best = float("inf")):
        dist = 0
        for i in range(n-1):
```

```
            city = path[i]
            city2 = path[i+1]
            dist += float(distances[(city, city2)])
            if dist > best:
                return              # quits when path is worse
        dist += float(distances[(city2, path[0])])
        return dist

# This function returns the best tour and its distance
# for a given number of cities n
def hill_climbing(n):
    nCities = cities[:n]        # global list

    # Swap 2 neighboring cities in given neighbor path
    def find_neighbor(nb):
        i = random.randint(0,n-2)
        j = i + 1
        nb[i], nb[j] = nb[j], nb[i]
        return nb

    best_tour = nCities
    random.shuffle(best_tour)
    best = get_distance(best_tour, n)
    nb_path = best_tour
    visited_neighbors = []

    i = 0
    while(i < 1000000):  #avoid infinite loops
        nb = find_neighbor(nb_path)
        # Avoid the visited neighbors
        if not nb_path in visited_neighbors:
            visited_neighbors.append(nb_path[:])    #adding copy
            nb_dist = get_distance(nb_path, n, best)

            #  Update best path if better
            if nb_dist != None:
                best = nb_dist
                best_tour = nb_path
            else:
                break           #if new path is not better
        i += 1

    return best_tour, best

print("HILL-CLIMBING FOR 10 CITIES")
start = time.time()
all_tours = {}
for i in range(20):
    tour, tour_d = hill_climbing(10)
    all_tours[tour_d] = tour
end = time.time()
HC_time = end - start

print("Executed time for 20 runs of the hill climbing method: ", "{:.10f}".format(
print("Compared to Exhaustive Search, this method performed" , "\033[1m{:.4f}\033[

keys = all_tours.keys()
best, worst, avg, std = min(keys), max(keys), sum(keys)/len(keys), np.std(list(key
best_tour = all_tours.get(best)
print("\nThe plot shows the best tour of the 20 runs: ", "{:.4f}".format(best), "\
plot_plan(best_tour)

print("The worst path of the 20 runs had a distance of : ", "{:.4f}".format(worst)
```

```
print("The mean: ", "{:.4f}".format(avg))
print("The standard deviation: ", "{:.4f}".format(std))

print("\nHILL-CLIMBING FOR 24 CITIES")
start = time.time()
all_tours = {}
for i in range(20):
    tour, tour_d = hill_climbing(24)
    all_tours[tour_d] = tour
end = time.time()
HC_time = end - start

print("Executed time for 20 runs of the hill climbing method: ", "{:.10f}".format(
print("Compared to Exhaustive Search, this method performed" , "\033[1m{:.4f}\033[


keys = all_tours.keys()
best, worst, avg, std = min(keys), max(keys), sum(keys)/len(keys), np.std(list(keys
best_tour = all_tours.get(best)
print("\nThe plot shows the best tour of the 20 runs: ", "{:.4f}".format(best), "\
plot_plan(best_tour)

print("The worst path of the 20 runs had a distance of : ", "{:.4f}".format(worst)
print("The mean: ", "{:.4f}".format(avg))
print("The standard deviation: ", "{:.4f}".format(std))
```

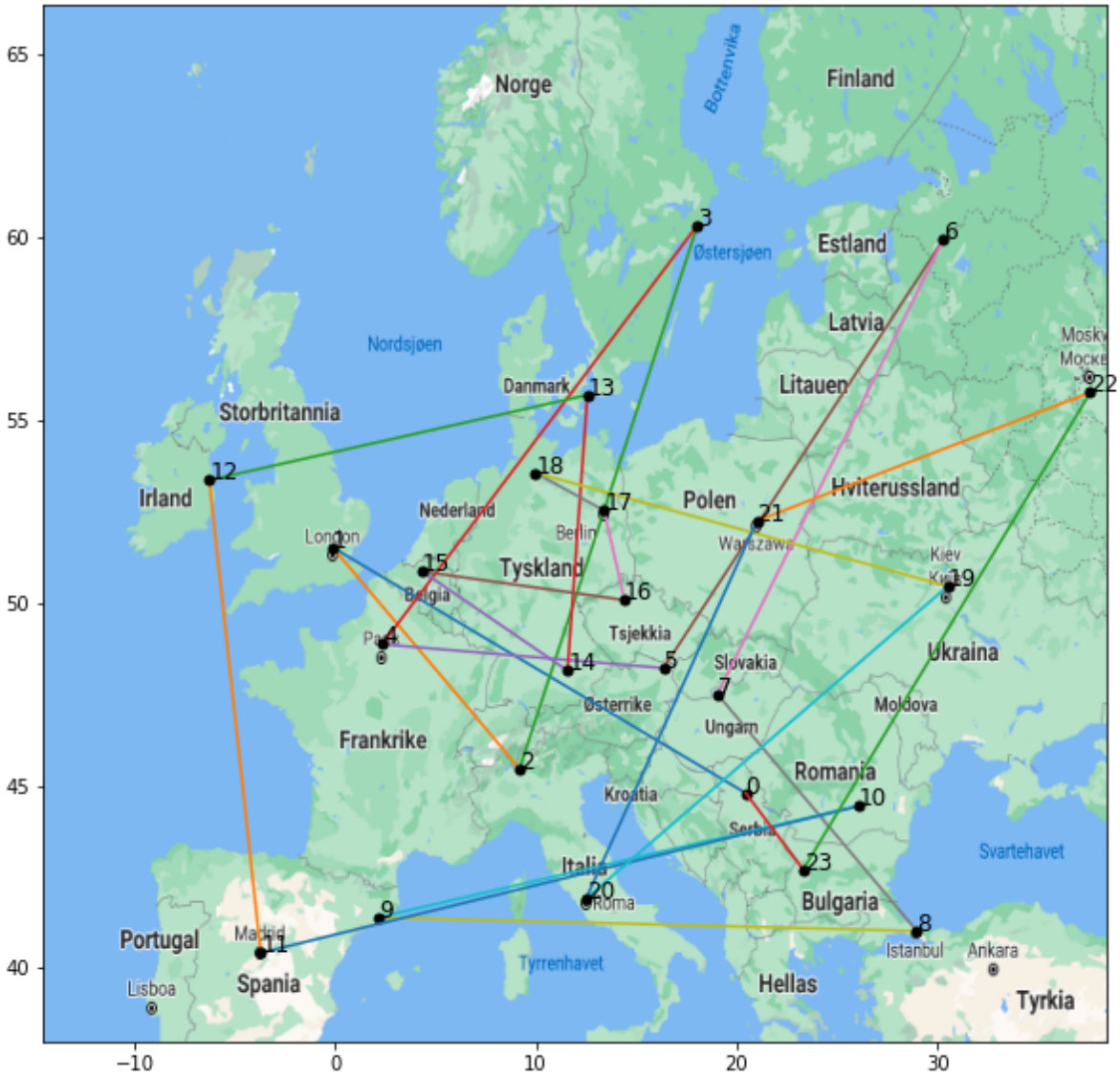
```
HILL-CLIMBING FOR 10 CITIES
Executed time for 20 runs of the hill climbing method:  0.0009448528
Compared to Exhaustive Search, this method performed 16.9540 s better!

The plot shows the best tour of the 20 runs:  9045.7700
With path :  ['Belgrade', 'Budapest', 'Barcelona', 'Hamburg', 'Brussels', 'Dubli
n', 'Bucharest', 'Berlin', 'Copenhagen', 'Istanbul']
```



The worst path of the 20 runs had a distance of : 13940.7300
The mean: 11374.1070
The standard deviation: 1272.9970

HILL-CLIMBING FOR 24 CITIES
Executed time for 20 runs of the hill climbing method: 0.0000000000
Compared to Exhaustive Search, this method performed **16.9549** s better!

The plot shows the best tour of the 20 runs: 27060.7500
With path : ['Belgrade', 'London', 'Milan', 'Stockholm', 'Paris', 'Vienna', 'Sain
t Petersburg', 'Budapest', 'Istanbul', 'Barcelona', 'Bucharest', 'Madrid', 'Dubli
n', 'Copenhagen', 'Munich', 'Brussels', 'Prague', 'Berlin', 'Hamburg', 'Kiev', 'Ro
me', 'Warsaw', 'Moscow', 'Sofia']




```
The worst path of the 20 runs had a distance of :  33851.6500
The mean:  30378.8270
The standard deviation:  2035.5661
```

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.

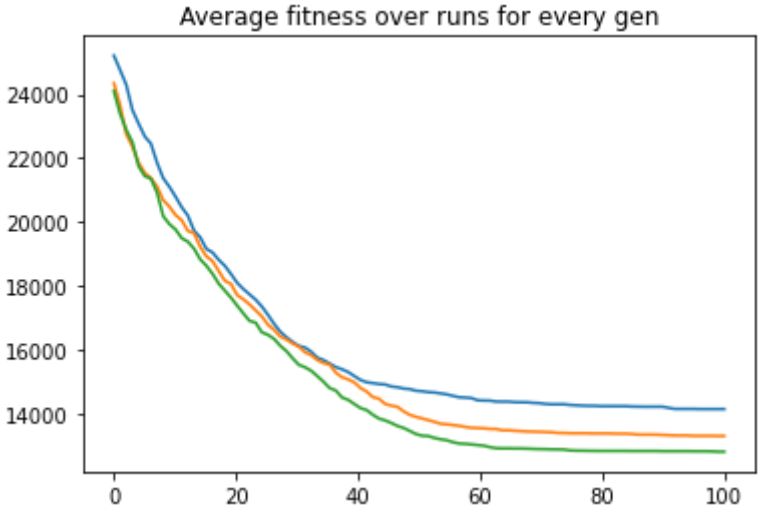
Finally, plot an example optimized tour (the best of the final generation) for the three different population sizes, using the plot_plan method.

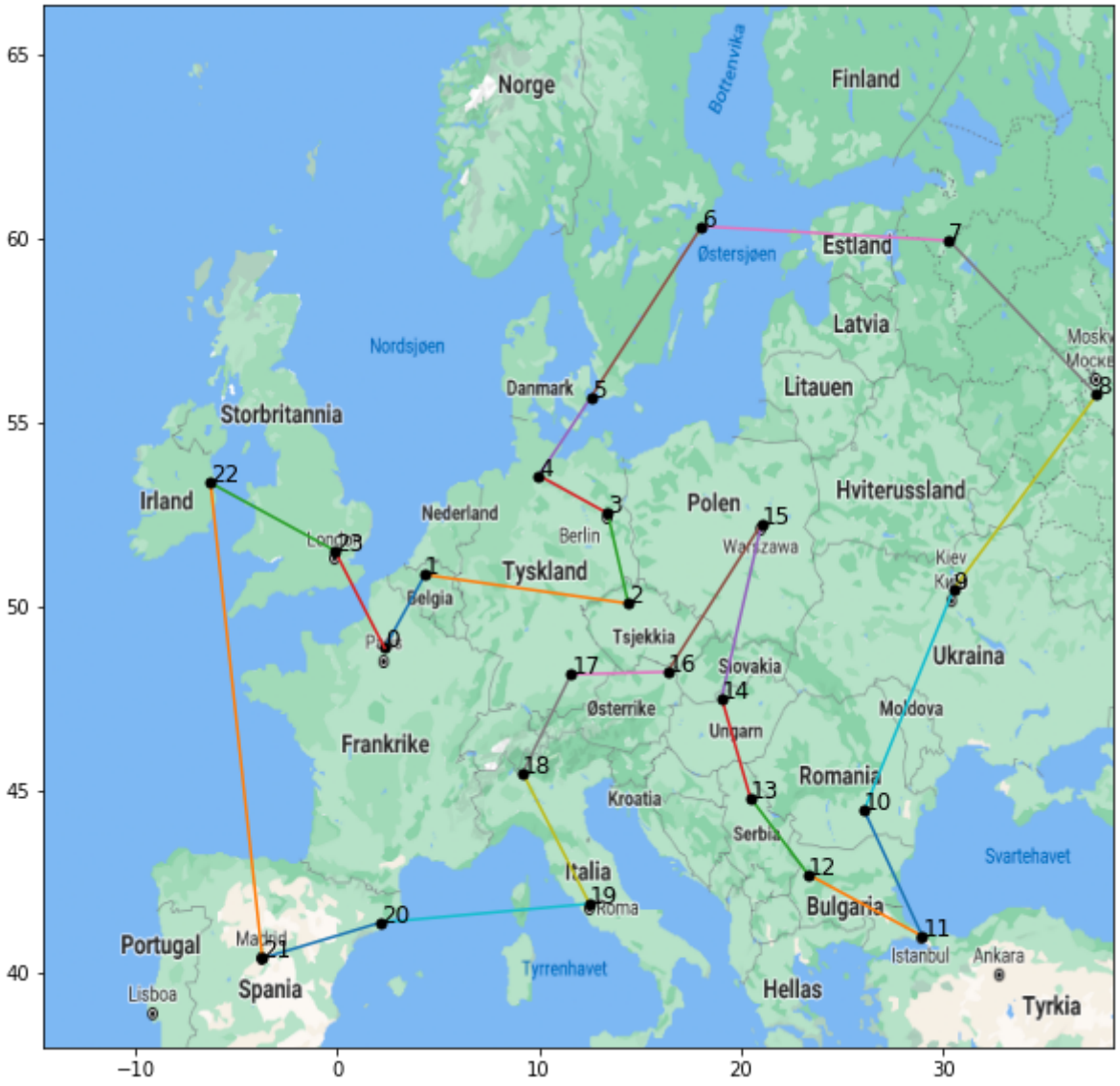
In [ ]:
```
# Implement the algorithm here
```

```
from collections import OrderedDict

def pmx_pair(p1, p2):
    l = len(p1)
    # choose segment
    left = random.randint(0,len(p1))
    right = random.randint(0,len(p1))
    if left > right:
        left, right = right, left

    # Initialize children with only the segments
    c1 = [None] * l
    c1[left:right] = p2[left:right]
    c2 = [None] * l
    c2[left:right] = p1[left:right]

    # Loop through parent 1
    for i in range(len(p1)):
        if not c1[i]:      # if the value is not yet assigned to the child 1
            j = p1[i]
            idx = None
            while j in c1:         # if the value is already in the child 1
                idx = c1.index(j)
                j = c2[idx]         # find the corresponding value from child 2
            c1[i] = j

    # Doing the same for parent 2 with child 2
    for i in range(len(p2)):
        if not c2[i]:
            j = p2[i]
            idx = None
            while j in c2:
                idx = c2.index(j)
                j = c1[idx]
            c2[i] = j

    return c1, c2

# Method for finding children with pmx
# returns a list of parents and children
def find_offsprings(sols):
    for i in range(0, len(sols), 2):
        child1, child2 = pmx_pair(sols[i], sols[i+1])
        if child1 not in sols:
            sols.append(child1)
        if child2 not in sols:
            sols.append(child2)
    return sols


def genetic(n, p, MAX, plotY):
    solutions = []
    lst = cities[:n].copy()
    # Initialization of n parents by random shuffling
    for i in range(p):
        random.shuffle(lst)
        solutions.append(lst[:])

    # Starting recombination for given generations
    gen = 0
    while (gen < MAX):
        solutions = find_offsprings(solutions)

        shortest_paths = sorted(solutions, key=lambda i: get_distance(i, n))   # s
```

```
        solutions = shortest_paths[:p]      # keep n best from this gen

        best_path = solutions[0]
        best = get_distance(best_path, n)

        plotY[gen] += best         # adds the best distance to the plot data
        gen += 1

    worst = get_distance(solutions[p-1], n)
    return plotY, (best, best_path), worst      # saves the best solution in tuple

# Running EA on n parents for certain generations
def runEA(n, p, gens):
    start = time.time()
    plotY = [0] * gens
    best_sols = []
    best, worst = (float("inf"), None), 0

    # Run 20 times
    for i in range(20):
        plotY, b, w = genetic(n, p, gens, plotY)
        # Find the best & worst
        if min(best[0], b[0]) == b[0]:
            best = b
        best_sols.append(best[0])
        worst = max(worst, w)

    # Finds the average fitness over 20 runs
    plotY = np.array(plotY)/20
    end = time.time()

    print("\nThe evolutionary algorithm for population size", p, "was performed in
    print("The best solution gives: ", round((best[0]), 3))
    print("The worst solution gives: ", round((worst), 3))
    print("The mean of the best solutions gives: ", round((np.mean(best_sols)), 3)
    print("The standard deviation of the best solutions gives: ", round((np.std(bes
    return plotY, best[1]      #returns the datapoints and the best path

gens = 100
pops = [100, 300, 500]
plt.title("Average fitness over runs for every gen")
plotX = np.linspace(0, gens, gens)      # x-axis

# Running EA for 24 cities on every population size
for i in range(len(pops)):
    plotY, best = runEA(24, pops[i], gens)
    pops[i] = best
    plt.plot(plotX, plotY)

# Showing plots of best solution of every population
for i in range(len(pops)):
    plot_plan(pops[i])
    print("This is the best solution for ", i+1, ". test with path: ", pops[i])

print("Conclusion: The best of them all is the test with size 300, because the best
print("Even though it finds a slightly better solution, it is at the expense of per
print("At an attempt on size 600 and additional 10s, it also didn't yield a signifi
```

```
The evolutionary algorithm for population size 100 was performed in  5.459 s.
The best solution gives:  12639.41
The worst solution gives:  16630.79
The mean of the best solutions gives:  13138.05
The standard deviation of the best solutions gives:  521.372

The evolutionary algorithm for population size 300 was performed in  18.761 s.
The best solution gives:  12504.65
The worst solution gives:  15140.46
The mean of the best solutions gives:  12575.126
The standard deviation of the best solutions gives:  56.328

The evolutionary algorithm for population size 500 was performed in  38.445 s.
The best solution gives:  12325.93
The worst solution gives:  14929.04
The mean of the best solutions gives:  12421.297
The standard deviation of the best solutions gives:  118.215
```





This is the best solution for 1 . test with path: ['Paris', 'Brussels', 'Prague', 'Berlin', 'Hamburg', 'Copenhagen', 'Stockholm', 'Saint Petersburg', 'Moscow', 'Kiev', 'Bucharest', 'Istanbul', 'Sofia', 'Belgrade', 'Budapest', 'Warsaw', 'Vienna', 'Munich', 'Milan', 'Rome', 'Barcelona', 'Madrid', 'Dublin', 'London']

This is the best solution for 2 . test with path: ['Saint Petersburg', 'Stockholm', 'Copenhagen', 'Hamburg', 'Brussels', 'Paris', 'London', 'Dublin', 'Madrid', 'Barcelona', 'Milan', 'Rome', 'Munich', 'Prague', 'Berlin', 'Warsaw', 'Vienna', 'Budapest', 'Belgrade', 'Sofia', 'Istanbul', 'Bucharest', 'Kiev', 'Moscow']


```
This is the best solution for  3 . test with path:  ['Sofia', 'Belgrade', 'Budapes
t', 'Vienna', 'Warsaw', 'Prague', 'Munich', 'Milan', 'Rome', 'Barcelona', 'Madri
d', 'Dublin', 'London', 'Paris', 'Brussels', 'Hamburg', 'Berlin', 'Copenhagen', 'S
tockholm', 'Saint Petersburg', 'Moscow', 'Kiev', 'Bucharest', 'Istanbul']
Conclusion: The best of them all is the test with size 300, because the best solut
ion isn't much different from size 500.
Even though it finds a slightly better solution, it is at the expense of performin
g time.
At an attempt on size 600 and additional 10s, it also didn't yield a significant i
mprovement
```

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?

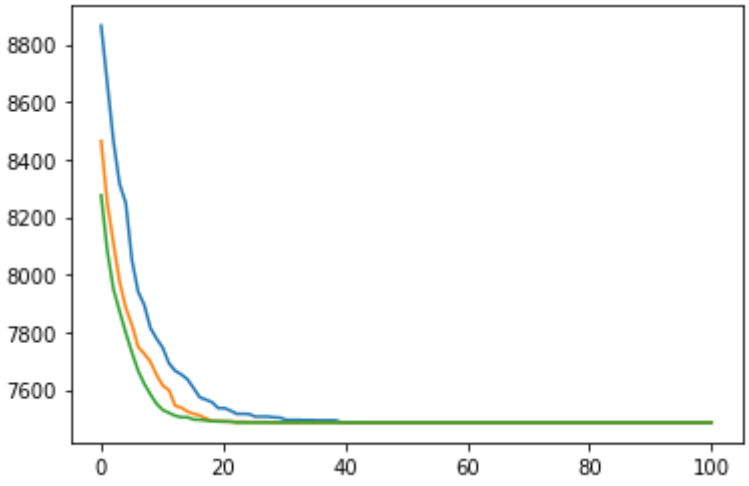
How many tours were inspected by your GA as compared to by the exhaustive search?

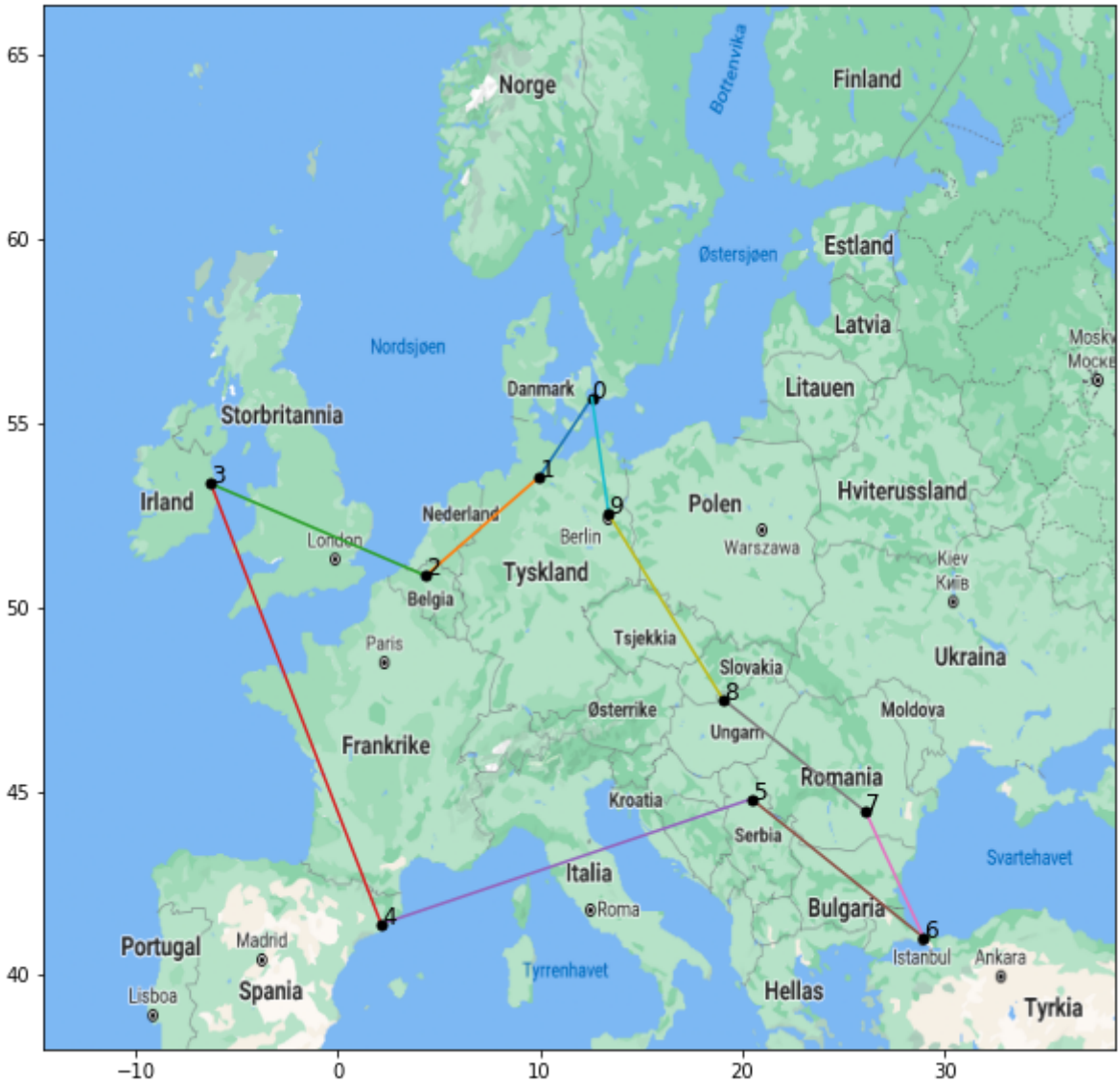
In [ ]:

```
# Answer
# Running EA for 10 cities on every population size
pops = [100, 300, 500]
for i in range(len(pops)):
    plotY, best = runEA(10, pops[i], gens)
    pops[i] = best
    plt.plot(plotX, plotY)
```

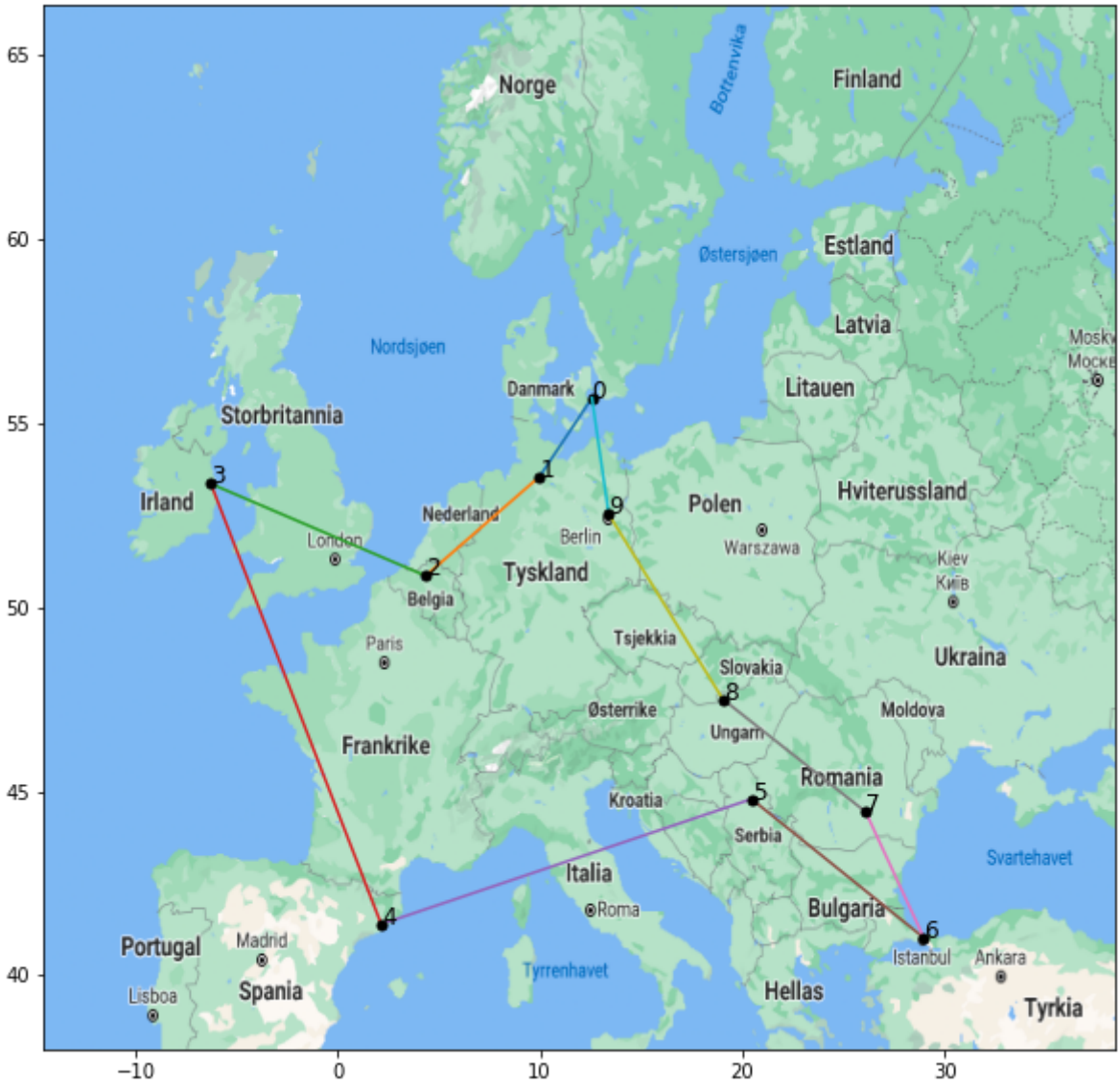
```
# Showing plots of best solution of every population
for i in range(len(pops)):
    plot_plan(pops[i])
    print("This is the best solution for ", i+1, ". test with path: ", pops[i])

print("My GA found the shortest tour for 10 cities with exact same distance.")
print("For 10 cities, the exhaustive search did better than mine with the populatio
print("The performing time was only 3s, 13s better!")
print("For 24 cities, the result always varied for everytime I tried running it. Th
print("at first attempt, but at the expense of a lot more time. Hence GA is much mo
```

```
The evolutionary algorithm for population size 100 was performed in  4.197 s.
The best solution gives:  7486.31
The worst solution gives:  8066.12
The mean of the best solutions gives:  7486.31
The standard deviation of the best solutions gives:  0.0

The evolutionary algorithm for population size 300 was performed in  11.705 s.
The best solution gives:  7486.31
The worst solution gives:  8009.86
The mean of the best solutions gives:  7486.31
The standard deviation of the best solutions gives:  0.0

The evolutionary algorithm for population size 500 was performed in  24.331 s.
The best solution gives:  7486.31
The worst solution gives:  8150.86
The mean of the best solutions gives:  7486.31
The standard deviation of the best solutions gives:  0.0
```

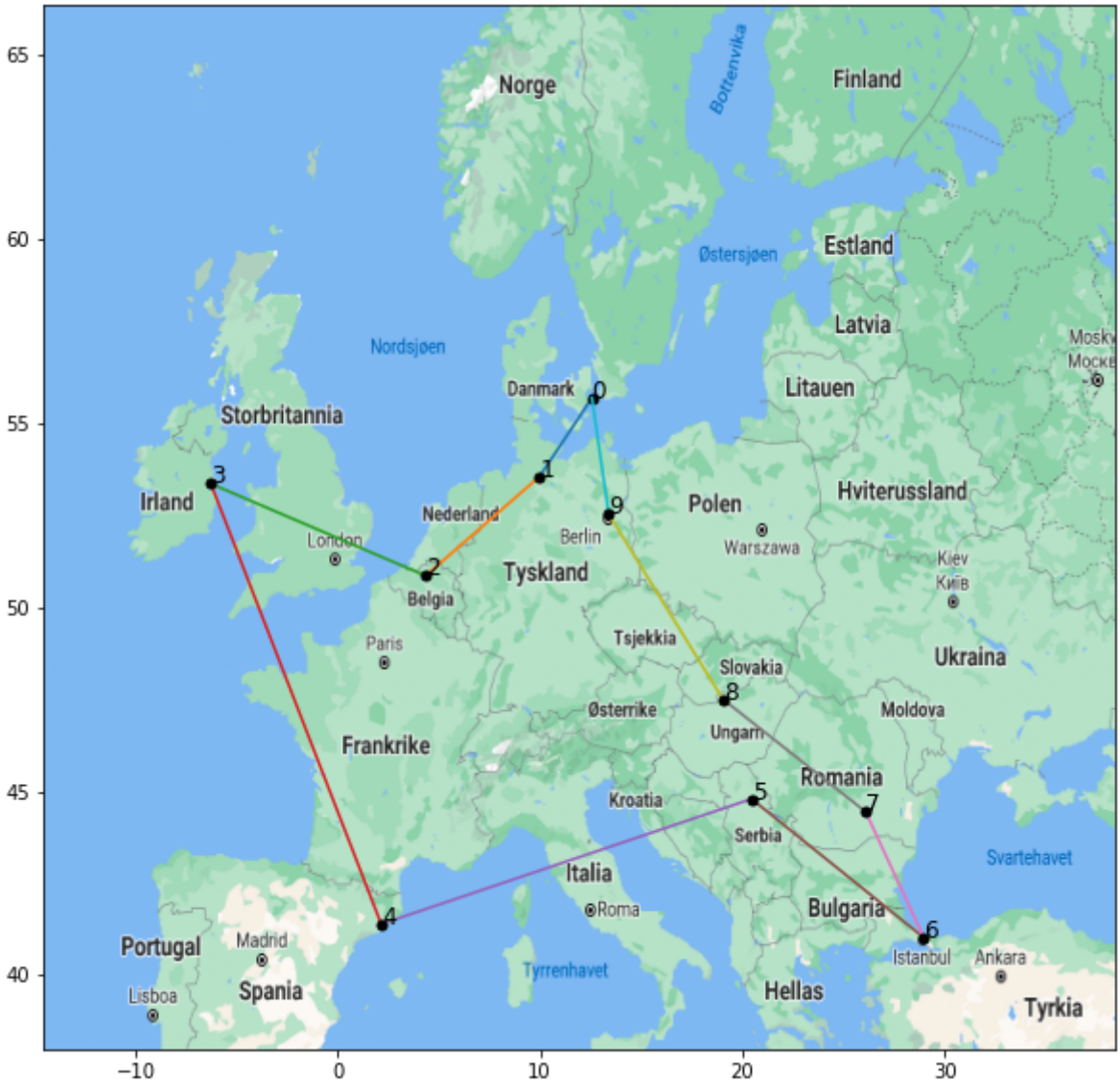





```
This is the best solution for  1 . test with path:  ['Copenhagen', 'Hamburg', 'Bru
ssels', 'Dublin', 'Barcelona', 'Belgrade', 'Istanbul', 'Bucharest', 'Budapest', 'B
erlin']
```




```
This is the best solution for  2 . test with path:  ['Copenhagen', 'Hamburg', 'Bru
ssels', 'Dublin', 'Barcelona', 'Belgrade', 'Istanbul', 'Bucharest', 'Budapest', 'B
erlin']
```




```
This is the best solution for  3 . test with path:  ['Copenhagen', 'Hamburg', 'Bru
ssels', 'Dublin', 'Barcelona', 'Belgrade', 'Istanbul', 'Bucharest', 'Budapest', 'B
erlin']
My GA found the shortest tour for 10 cities with exact same distance.
For 10 cities, the exhaustive search did better than mine with the population size
500,
 but size 100 was more than good enough to find the best solution.
The performing time was only 3s, 13s better!
For 24 cities, the result always varied for everytime I tried running it. Therefor
e exhaustive search would have found the best solution
at first attempt, but at the expense of a lot more time. Hence GA is much more eff
ective, but not always optimal to find the best solution.
```

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 [ ]:
```
# Implement algorithm here
```