Assignment1

March 10, 2023

1 IN3050/IN4050 Mandatory Assignment 1: Traveling Salesman Problem

1.1 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.

1.1.1 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.

1.1.2 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.

1.2 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.

1.3 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)

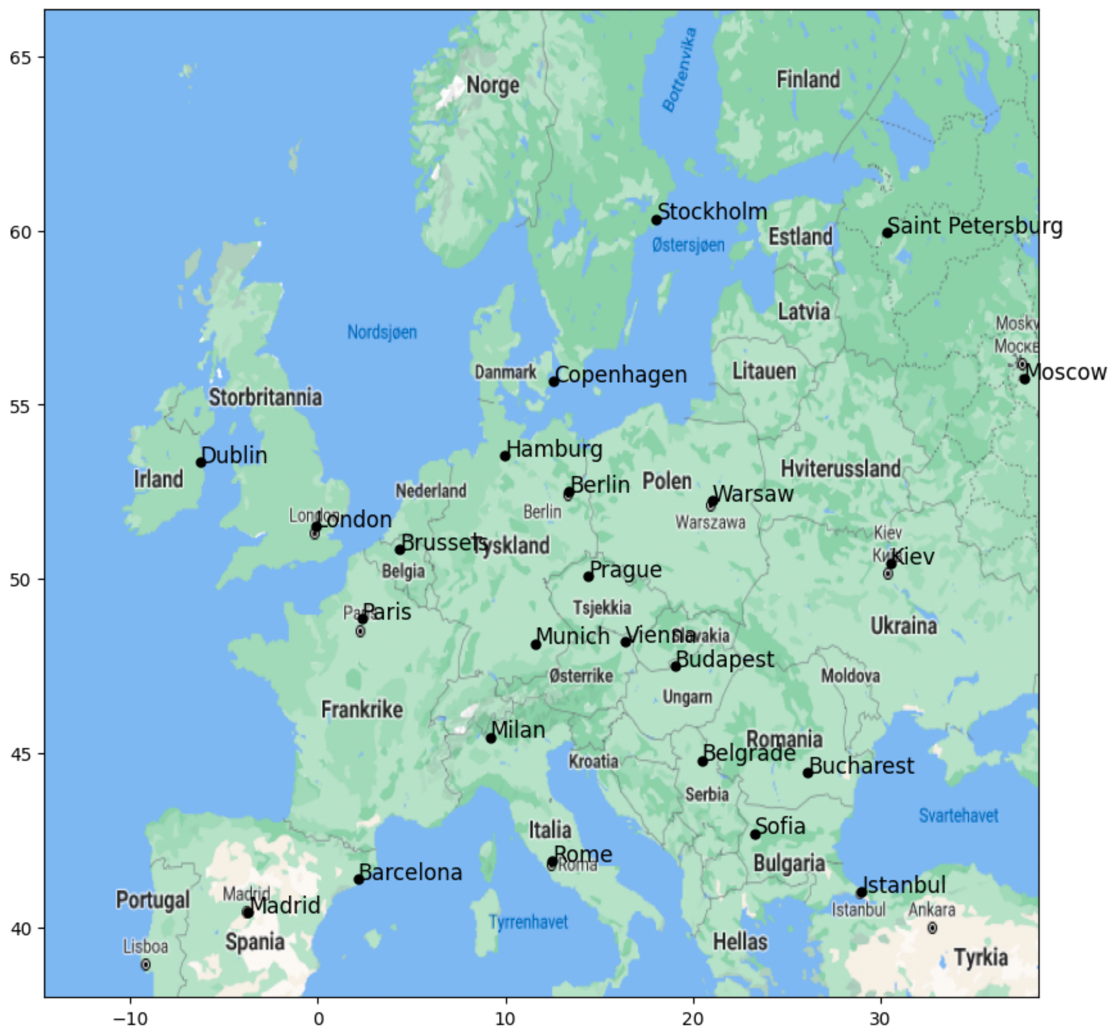
1.4 Helper code for visualizing solutions

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.

[ ]:
```
%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": [13.40,52.52], "Brussels":[4.35,50.85],"Bucharest":[26.10,44.44],␣
↪"Budapest": [19.04,47.50], "Copenhagen":[12.57,55.68], "Dublin":[-6.27,53.
↪35], "Hamburg": [9.99, 53.55], "Istanbul": [28.98, 41.02], "Kiev": [30.52,50.
↪45], "London": [-0.12,51.51], "Madrid": [-3.70,40.42], "Milan":[9.19,45.46],␣
↪"Moscow": [37.62,55.75], "Munich": [11.58,48.14], "Paris":[2.35,48.86],␣
↪"Prague":[14.42,50.07], "Rome": [12.50,41.90], "Saint Petersburg": [30.31,59.
↪94], "Sofia":[23.32,42.70], "Stockholm": [18.06,60.33],"Vienna":[16.36,48.
↪21],"Warsaw":[21.02,52.24]}
```

```
[ ]: #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␣
↪= "auto")



# 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);
```

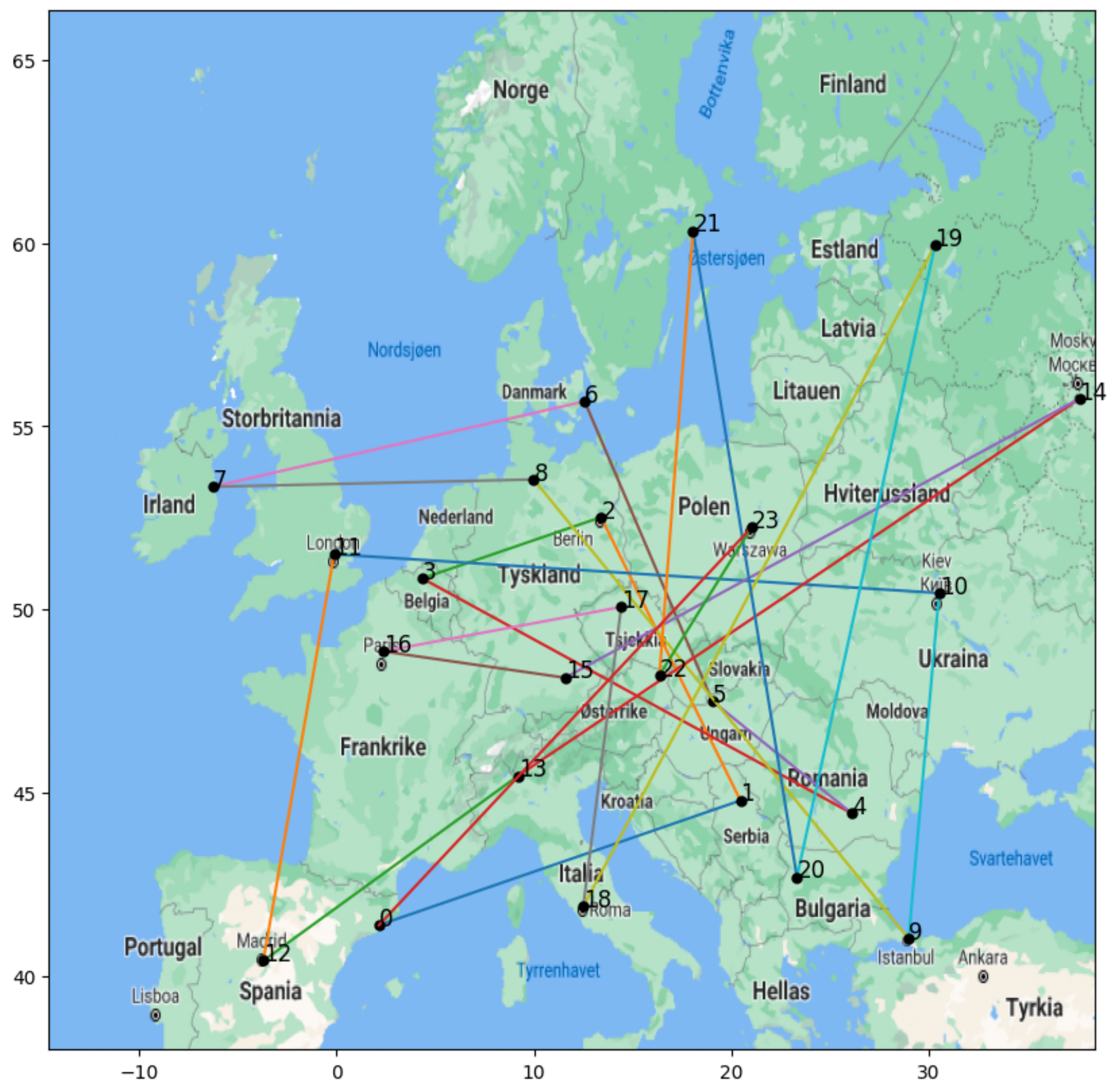


```
[ ]: #A method you can use to plot your plan on the map.
     def plot_plan(city_order):
         fig, ax = plt.subplots(figsize=(10,10))
         ax.imshow(europe_map, extent=[-14.56,38.43, 37.697 +0.3 , 64.344 +2.0],␣
      ↪aspect = "auto")

         # 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();
```

[ ]:
```
#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',
'Copenhagen', 'Dublin', 'Hamburg', 'Istanbul', 'Kiev', 'London', 'Madrid',
'Milan', 'Moscow', 'Munich', 'Paris', 'Prague', 'Rome', 'Saint Petersburg',
'Sofia', 'Stockholm', 'Vienna', 'Warsaw']
```



1.5 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.

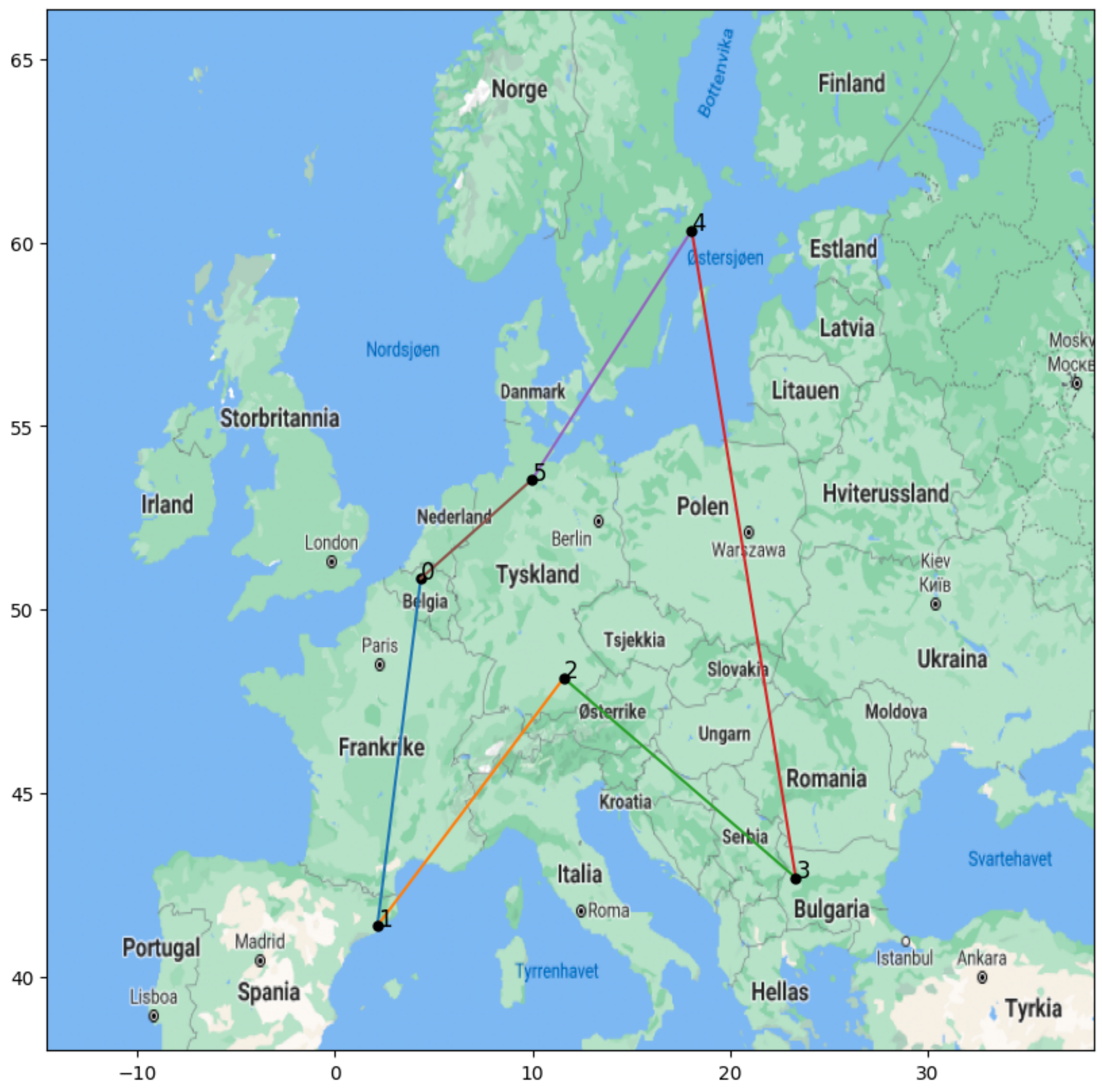
```
[ ]: # Implement the algorithm here
     import itertools as it
     import networkx as nx
     import random

     def sum_weights(plan, graph):
         return sum([graph[plan[i-1]][plan[i]]['weight'] for i in range(len(plan))])
```

```
def exhaustive_search(graph: nx.Graph):
    nodes = list(graph.nodes)
    res   = (nodes, sum_weights(nodes, graph))
    pmts  = it.permutations(nodes)

    for pmt in pmts:
        new_sum = sum_weights(pmt, graph)
        old_sum = sum_weights(res[0], graph)

        if new_sum < old_sum:
            res = (pmt, new_sum)

    return res
```

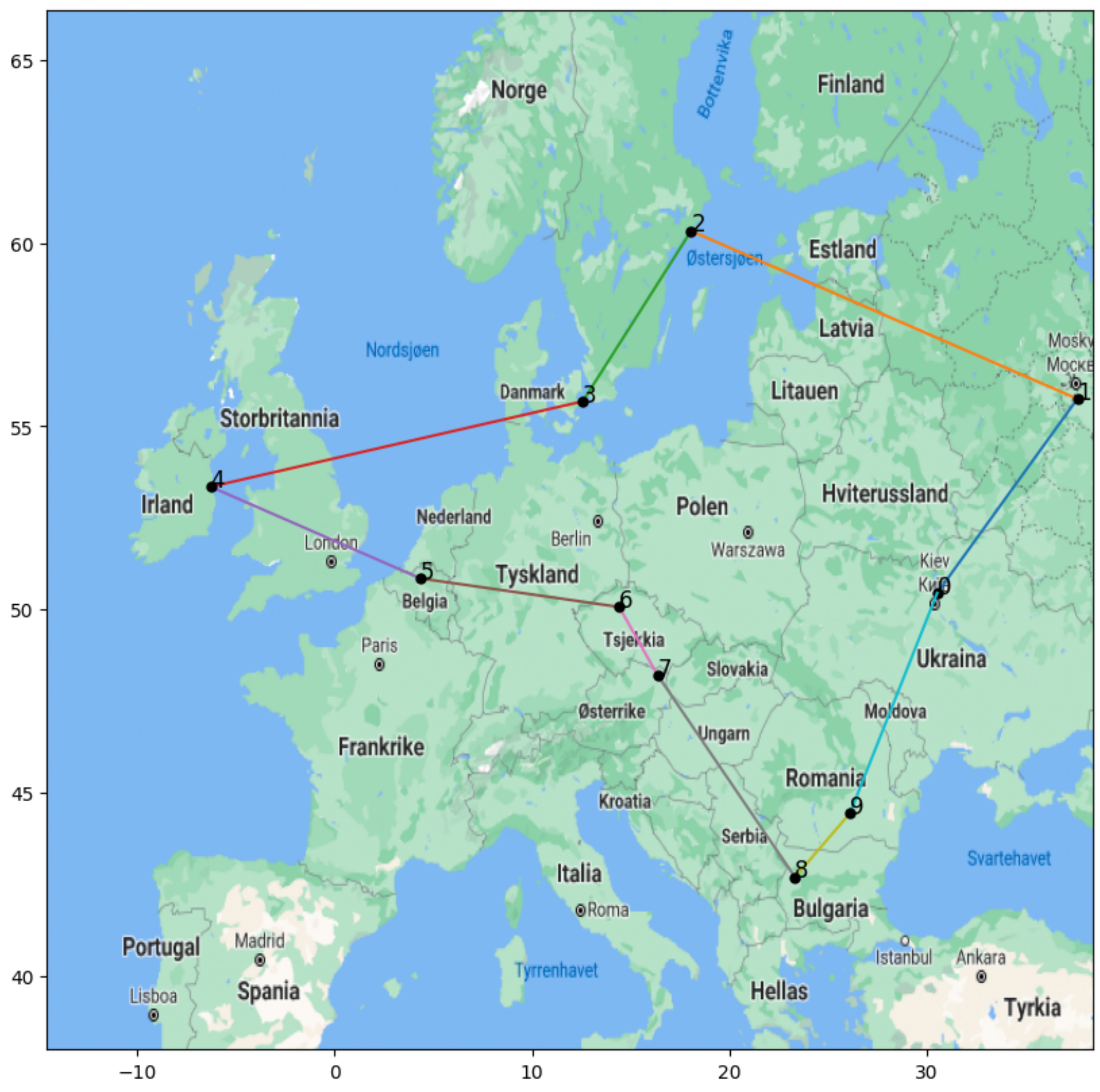
[ ]:
```
import time

G = nx.Graph()
for node_from in cities:
    G.add_node(node_from)
    for node_to in cities:
        G.add_edge(node_from, node_to, weight=float(data[cities.index(node_to)␣
 ↪+ 1][cities.index(node_from)]))

subset6 = random.sample(list(G.nodes), 6)
subset10 = random.sample(list(G.nodes), 10)
subset24 = random.sample(list(G.nodes), 24)

sub6 = G.subgraph(subset6)
sub10 = G.subgraph(subset10)
sub24 = G.subgraph(subset24)
```

[ ]:
```
start = time.time()
plan_ex6 = exhaustive_search(sub6)
end = time.time()

print("TIME: ", end - start)

print(plan_ex6)
plot_plan(plan_ex6[0])

start = time.time()
plan_ex10 = exhaustive_search(sub10)
end = time.time()

print("TIME: ", end - start)
print(plan_ex10)
plot_plan(plan_ex10[0])
```

```
TIME:  0.041887521743774414
(('Brussels', 'Barcelona', 'Munich', 'Sofia', 'Stockholm', 'Hamburg'),
6398.299999999999)
```



```
TIME:  175.68974804878235
(('Kiev', 'Moscow', 'Stockholm', 'Copenhagen', 'Dublin', 'Brussels', 'Prague',
'Vienna', 'Sofia', 'Bucharest'), 7347.61)
```



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?

1.6 Answer

The travelling salesman problem in NP-Hard and O(n!) so, given that the difference between 6 and 10 cities is already a factor of 3 with 24 cities it would take for my computer approximately

$10! : 3 = 24! : x$

$x = 5.1 * 10^{17}$ minutes $\approx$ 1000 billion years

1.7 Hill Climbing

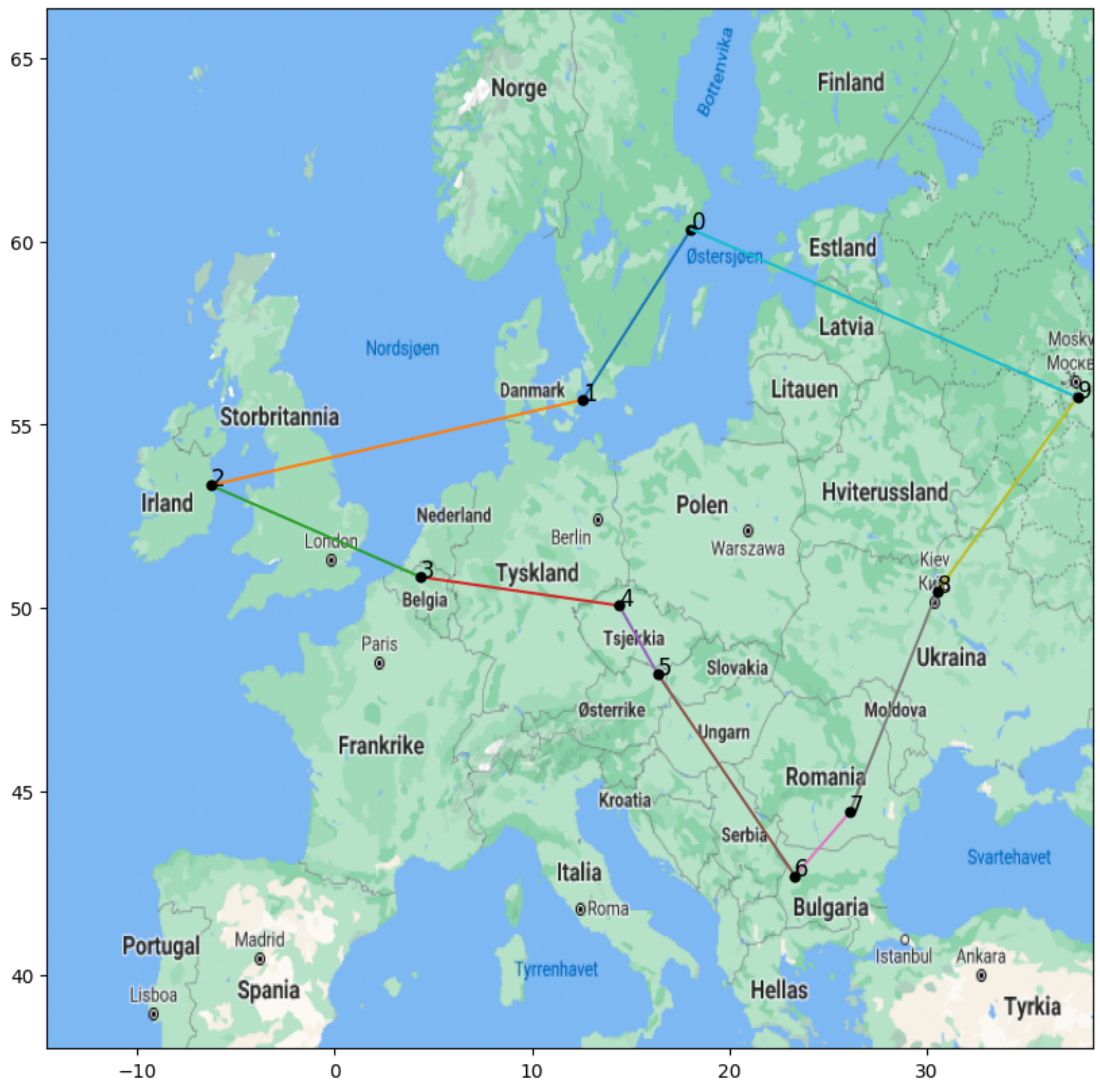
Then, write a simple hill climber to solve the TSP. How well does the hill climber perform, compared to the result from the exhaustive search for the first **10 cities**? Since you are dealing with a stochastic algorithm, you should run the algorithm several times to measure its performance. Report the 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).

[ ]:
```python
# Implement the algorithm here
import numpy as np

def get_neighbours(plan):
    neighbours = []
    for i in range(len(plan)):
        for j in range(i+1, len(plan)):
            new_plan = plan.copy()
            new_plan[i], new_plan[j] = new_plan[j], new_plan[i]
            neighbours.append(new_plan)

    return neighbours

def map_neighbours(plan, graph):
    return [(neighbour, sum_weights(neighbour, graph)) for neighbour in␣
 ↪get_neighbours(plan)]

def hill_climbing(graph: nx.Graph):
    nodes = list(graph.nodes)
    start = np.random.permutation(nodes)
    res   = (start, sum_weights(start, graph))

    new_route, new_route_length  = min(map_neighbours(res[0], graph),␣
 ↪key=lambda x: x[1])

    while new_route_length < res[1]:
        res = (new_route, new_route_length)
        new_route, new_route_length  = min(map_neighbours(res[0], graph),␣
 ↪key=lambda x: x[1])

    return res
```

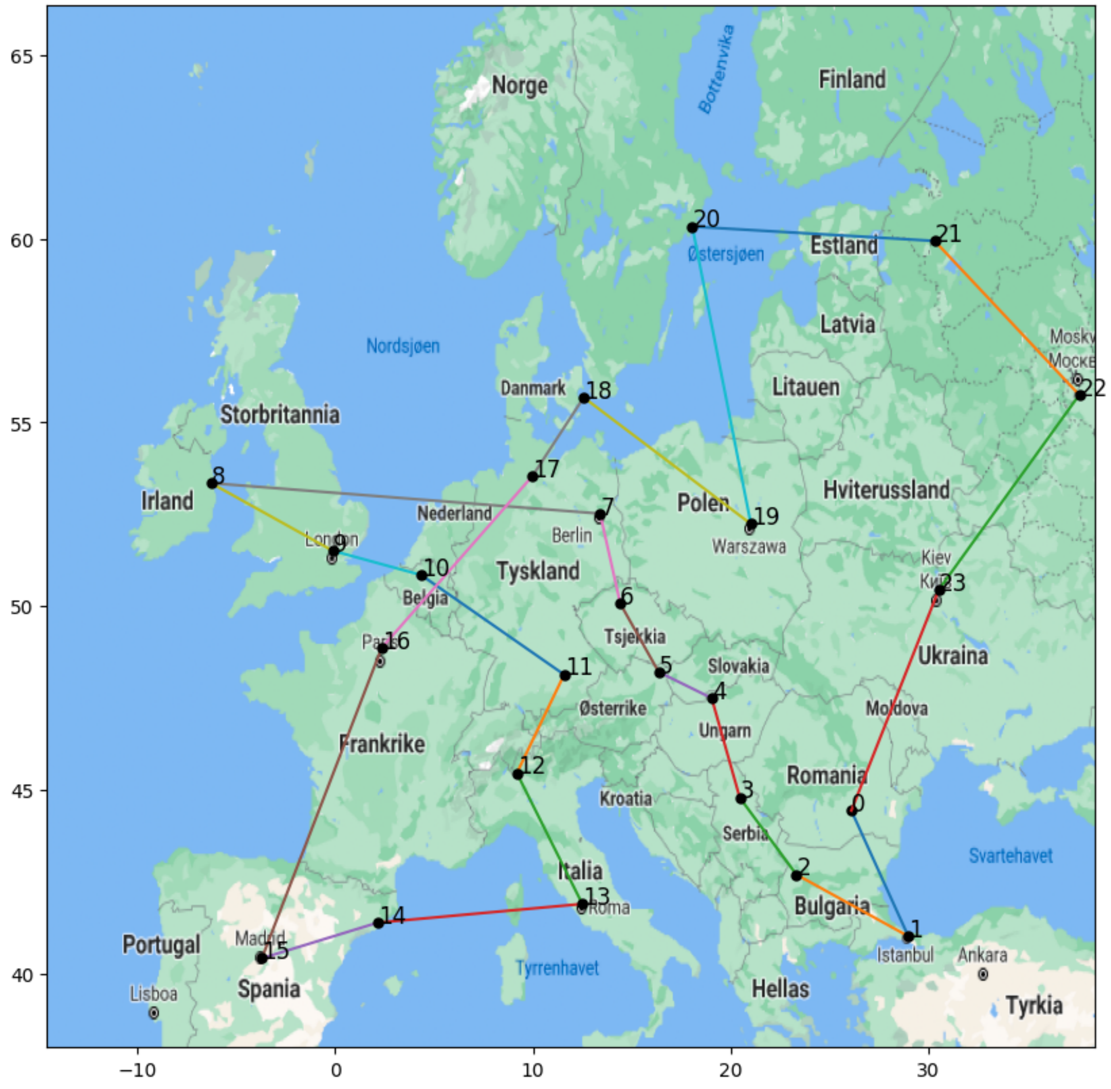
[ ]:
```python
start = time.time()
plan_hc10 = hill_climbing(sub10)
end = time.time()

print("TIME: ", end - start)

print(plan_hc10)
```

```
plot_plan(plan_hc10[0])

start = time.time()
plan_hc24 = hill_climbing(sub24)
end = time.time()

print("TIME: ", end - start)

print(plan_hc24)
plot_plan(plan_hc24[0])
```

```
TIME:  0.0655362606048584
(array(['Stockholm', 'Copenhagen', 'Dublin', 'Brussels', 'Prague',
       'Vienna', 'Sofia', 'Bucharest', 'Kiev', 'Moscow'], dtype='<U10'),
7347.61)
```

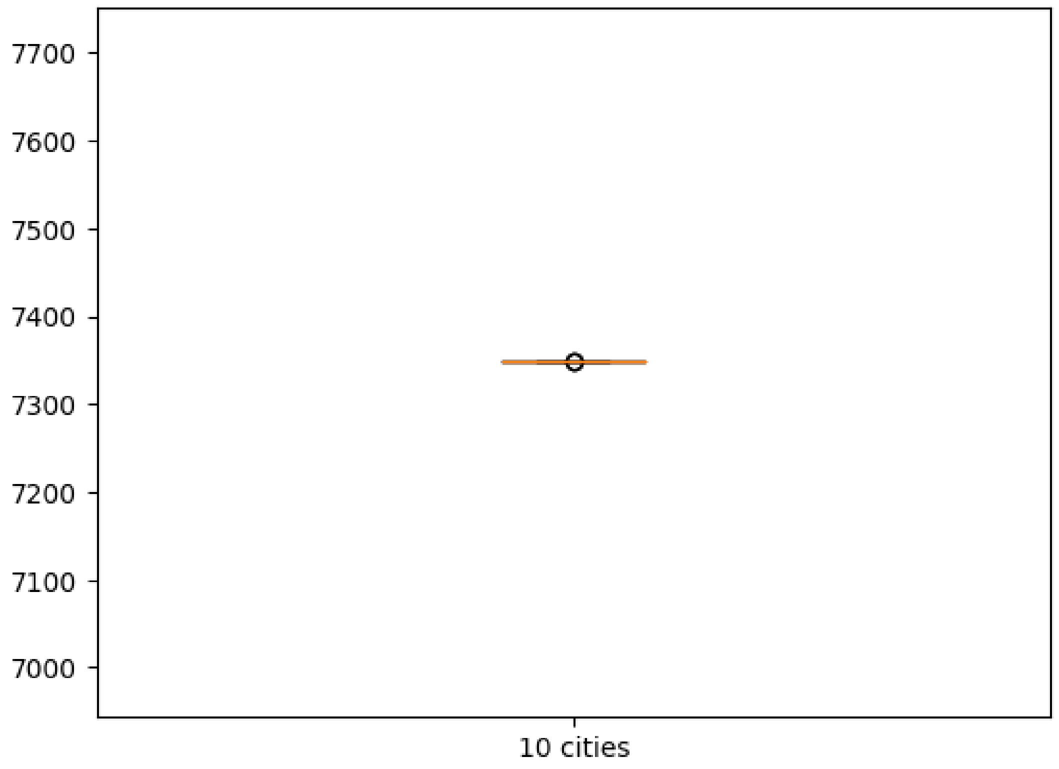


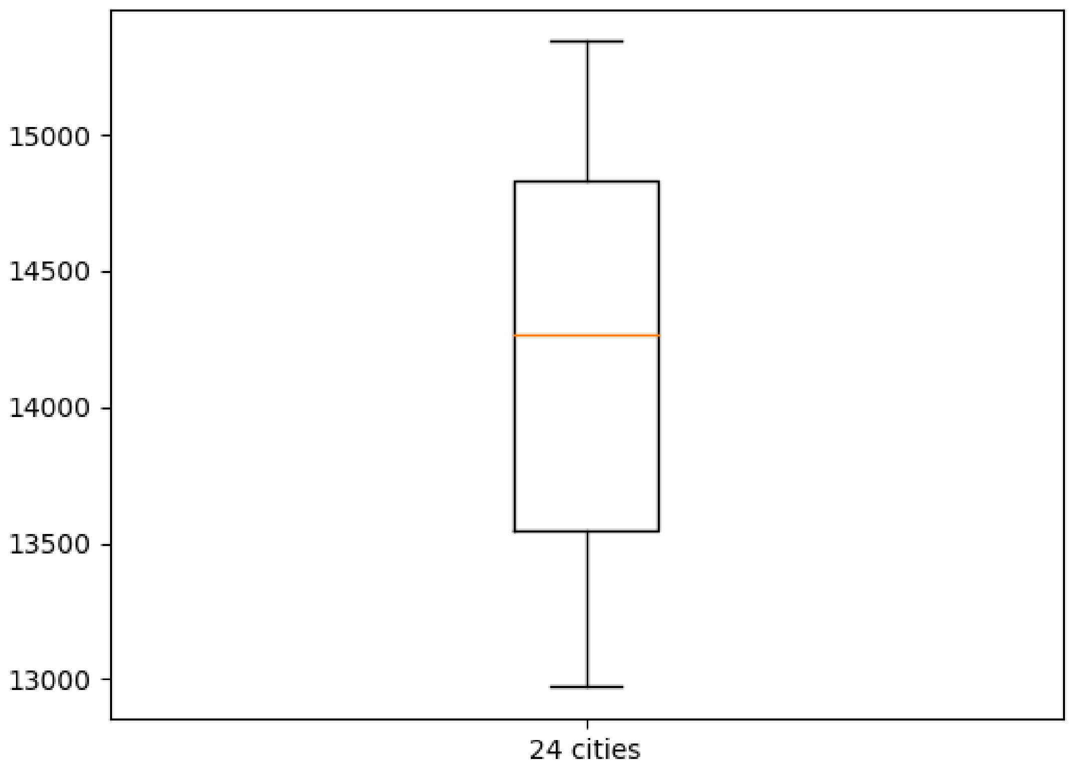
```
TIME:  0.5268440246582031
(array(['Bucharest', 'Istanbul', 'Sofia', 'Belgrade', 'Budapest', 'Vienna',
       'Prague', 'Berlin', 'Dublin', 'London', 'Brussels', 'Munich',
       'Milan', 'Rome', 'Barcelona', 'Madrid', 'Paris', 'Hamburg',
       'Copenhagen', 'Warsaw', 'Stockholm', 'Saint Petersburg', 'Moscow',
       'Kiev'], dtype='<U16'), 13610.61)
```



```
[ ]: import matplotlib.pyplot as plt

     res10 = []
```

```
for i in range(20):
    res10.append(hill_climbing(sub10))

fig = plt.figure()
plt.boxplot([x[1] for x in res10], labels=['10 cities'])
plt.show()

res24 = []

for i in range(20):
    res24.append(hill_climbing(sub24))

fig = plt.figure()
plt.boxplot([x[1] for x in res24], labels=['24 cities'])
plt.show()
```





1.8 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).

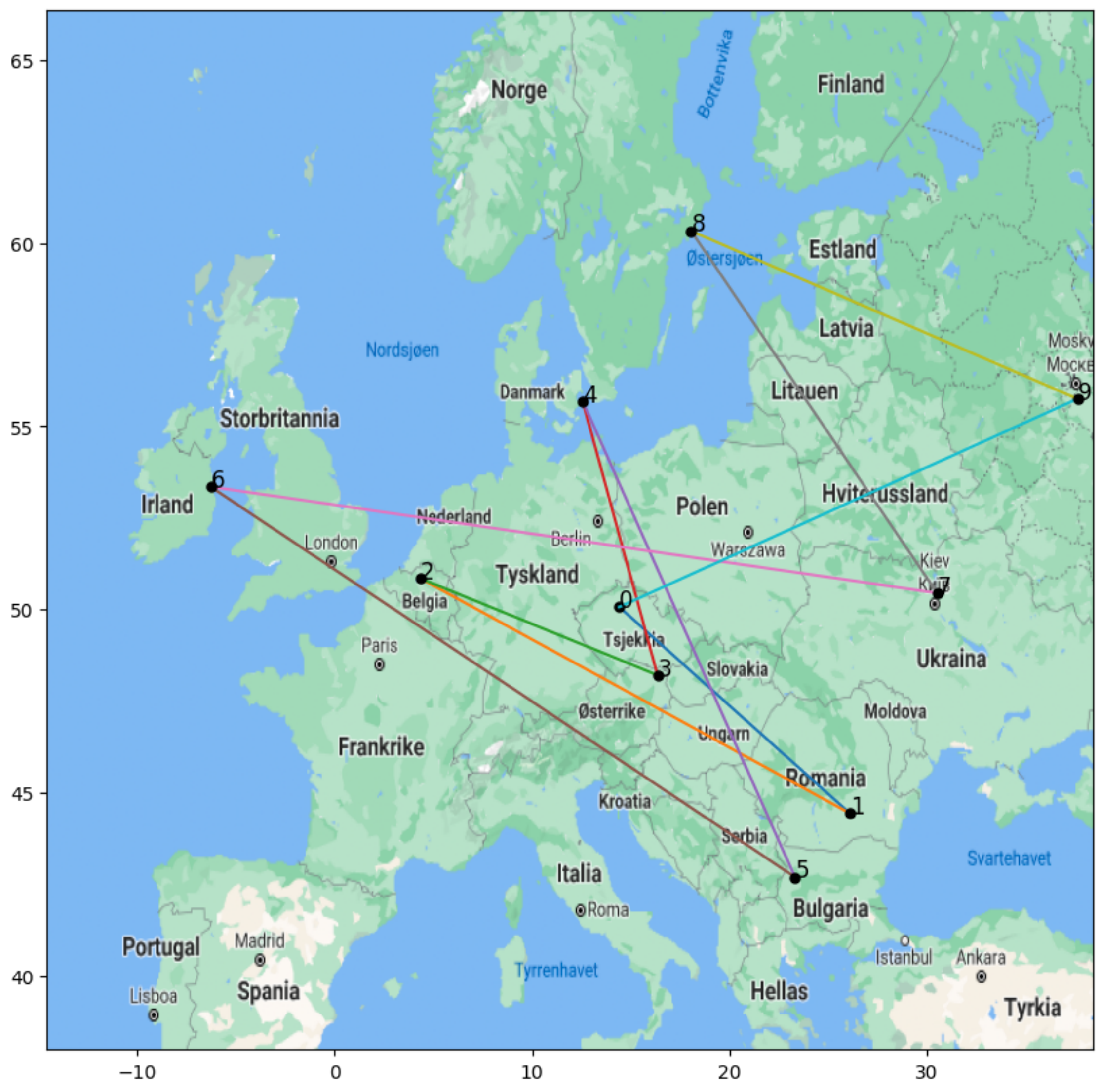
For all three variants: As with the hill climber, report best, worst, mean and standard deviation of tour length out of 20 runs of the algorithm (of the best individual of last generation). Also, find and plot the average fitness of the best fit individual in each generation (average across runs), and 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.

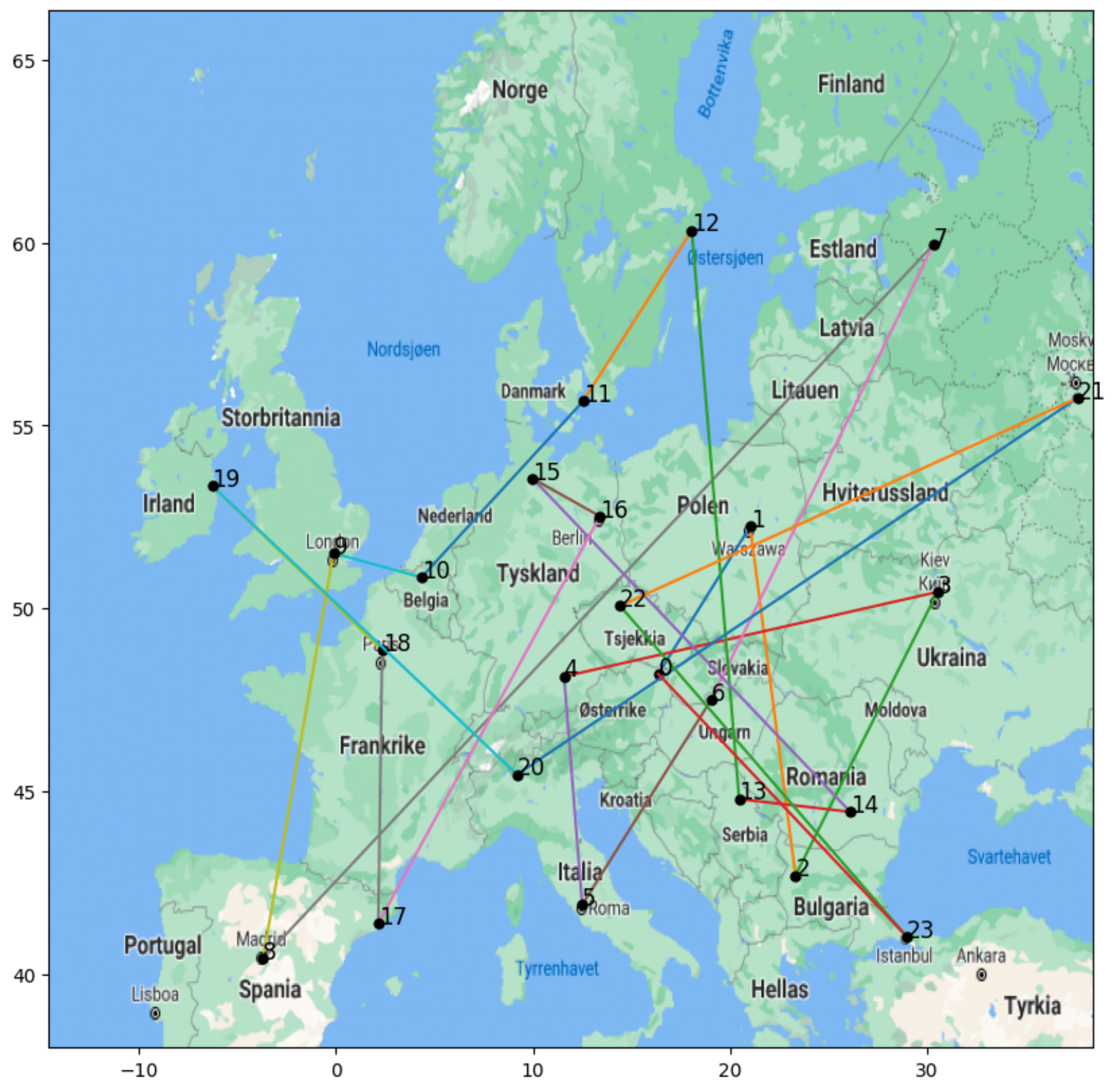
[ ]:
```python
# Implement the algorithm here
import bisect

# Mutation function which randomply swaps two cities in the plan
def mutate(plan):
    i = random.randint(0, len(plan) - 1)
    j = random.randint(0, len(plan) - 1)
    plan[i], plan[j] = plan[j], plan[i]

    return plan

# Simple implemenetation of tournament selection, select mu_percentage of the
# population twice and return the best mu_percentage of the population
def select_parents(population, mu_percentage):
    n = len(population) * mu_percentage // 100
    first_selection  = random.sample(population, n)
    second_selection = random.sample(population, n)

    return sorted(first_selection + second_selection, key=lambda x: x[1])[:n]

# Simple ordered crossover function which swaps random sequences of cities␣
 ↪between two plans
def pmx(p1, p2):
    n = len(p1)
    i, j = sorted(random.sample(range(n), 2))

    child = [None] * n
    child[i:j] = p1[i:j]

    for k in range(n):
        if k not in range(i, j):
            value = p2[k]
            while value in child:
                index = p1.index(value)
                value = p2[index]
            child[k] = value

    return child

def genetic_algorithm(graph: nx.Graph, population_size=500,␣
 ↪mutation_probability=0.20, mu_percentage=30, iterations=2000):
    nodes = list(graph.nodes)
    best_fit_individuals = []

    population = []
    for i in range(population_size):
        plan = np.random.permutation(nodes)
        individual = (plan, sum_weights(plan, graph))
        bisect.insort(population, individual, key=lambda x: x[1])

    best_fit_individuals.append(population[0][1])

    while iterations > 0:
        parents = select_parents(population, mu_percentage)

        for p in parents:
            if random.random() < mutation_probability:
                mutation = mutate(p[0])
                bisect.insort(population, (mutation, sum_weights(mutation,␣
↪graph)), key=lambda x: x[1])
            else:
                # Select a second parent, possible to cross with itself, can be␣
↪improved
                x = random.randint(0, len(parents) - 1)
                child = pmx(list(p[0]), list(parents[x][0]))
                bisect.insort(population, (child, sum_weights(child, graph)),␣
↪key=lambda x: x[1])

        population = population[:population_size]
        best_fit_individuals.append(population[0][1])
        iterations -= 1

    return population[0], best_fit_individuals
```

[ ]:
```
start = time.time()
plan_ga10, best_fit_individuals10 = genetic_algorithm(sub10)
end = time.time()

print("TIME: ", end - start)

print(plan_ga10)
plot_plan(plan_ga10[0])

start = time.time()
plan_ga24, best_fit_individuals24 = genetic_algorithm(sub24)
end = time.time()

print("TIME: ", end - start)

print(plan_ga24)
plot_plan(plan_ga24[0])
```

```
TIME:  15.401881694793701
(['Prague', 'Bucharest', 'Brussels', 'Vienna', 'Copenhagen', 'Sofia', 'Dublin',
'Kiev', 'Stockholm', 'Moscow'], 7347.61)
```



```
TIME:  33.012089014053345
(['Vienna', 'Warsaw', 'Sofia', 'Kiev', 'Munich', 'Rome', 'Budapest', 'Saint
Petersburg', 'Madrid', 'London', 'Brussels', 'Copenhagen', 'Stockholm',
'Belgrade', 'Bucharest', 'Hamburg', 'Berlin', 'Barcelona', 'Paris', 'Dublin',
'Milan', 'Moscow', 'Prague', 'Istanbul'], 12340.500000000002)
```

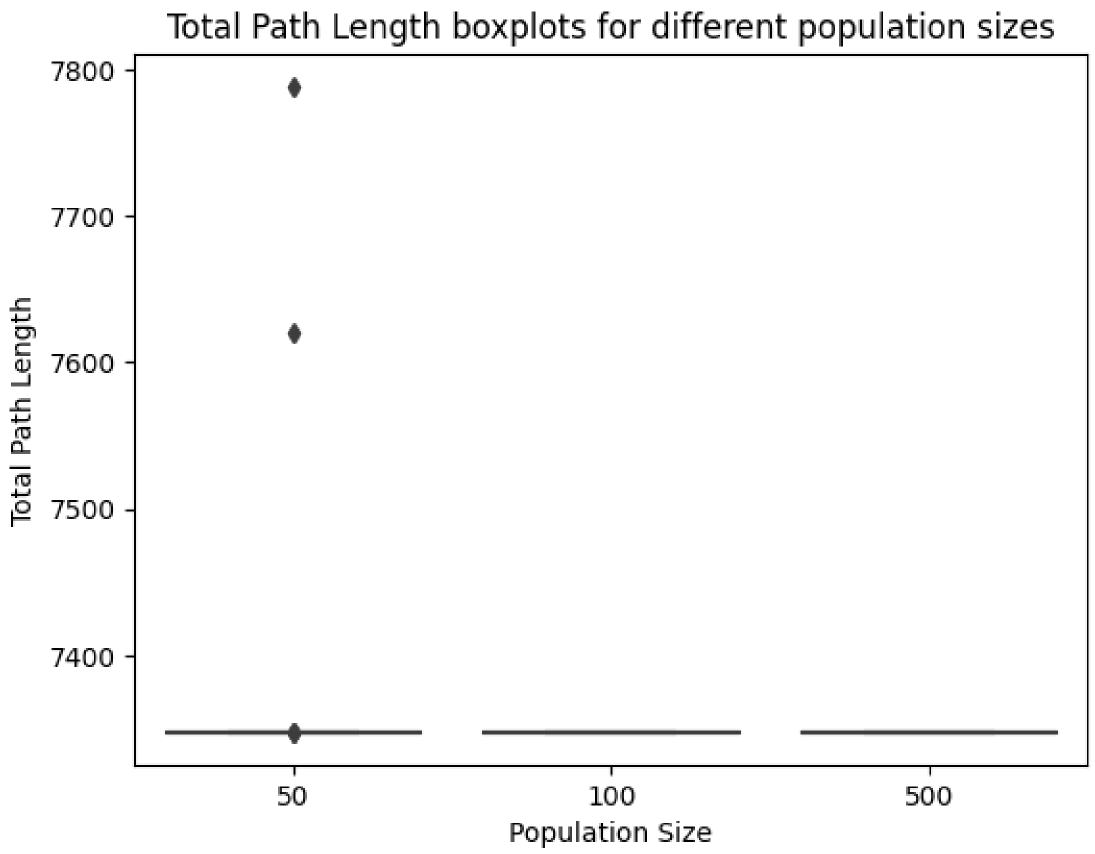


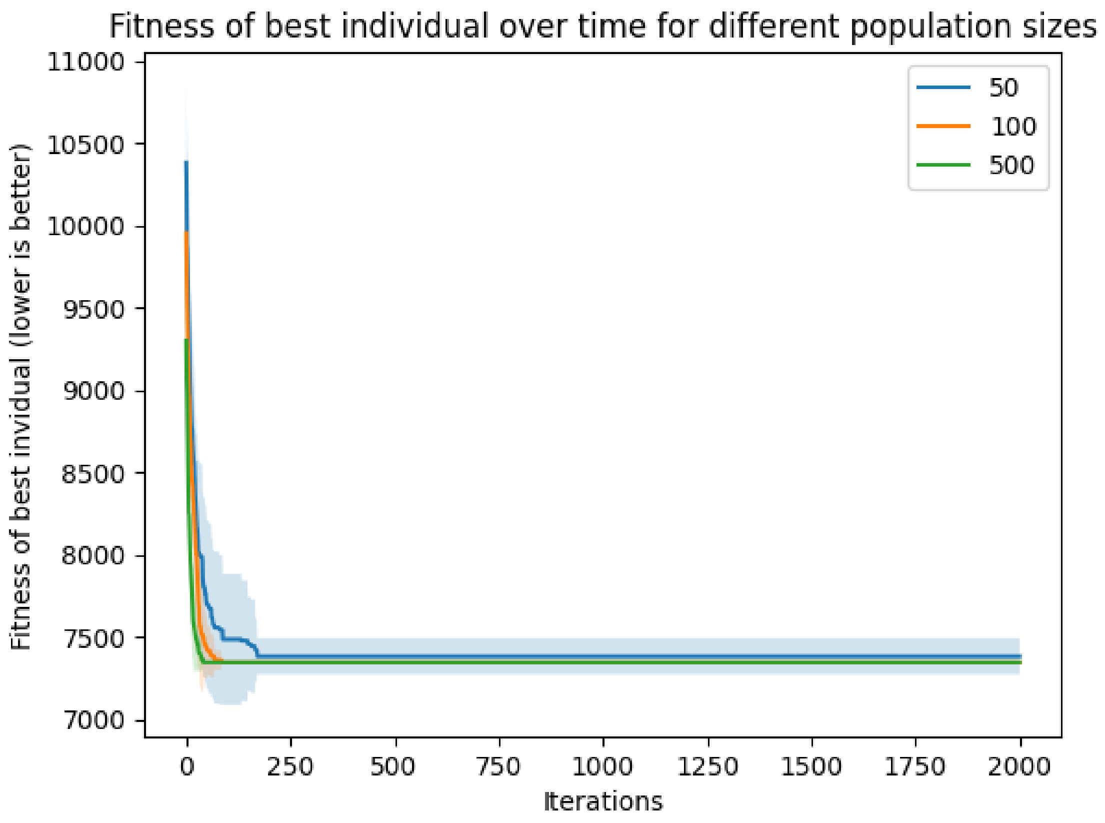
```
[ ]: res10_pop50  = []
     res10_pop100 = []
     res10_pop500 = []

     # 2000 + 1 iterations, can be extracted to constant, would be better
     res10_fit_pop50  = np.zeros((20, 2001))
     res10_fit_pop100 = np.zeros((20, 2001))
     res10_fit_pop500 = np.zeros((20, 2001))

     for i in range(20):
         plan_ga10_pop50, best_fit_individuals10_pop50 = genetic_algorithm(sub10,␣
      ↪population_size=50)
```

```
    plan_ga10_pop100, best_fit_individuals10_pop100 = genetic_algorithm(sub10,␣
 ↪population_size=100)
    plan_ga10_pop500, best_fit_individuals10_pop500 = genetic_algorithm(sub10,␣
 ↪population_size=500)

    res10_pop50.append(plan_ga10_pop50)
    res10_pop100.append(plan_ga10_pop100)
    res10_pop500.append(plan_ga10_pop500)

    res10_fit_pop50[i]  = best_fit_individuals10_pop50
    res10_fit_pop100[i] = best_fit_individuals10_pop100
    res10_fit_pop500[i] = best_fit_individuals10_pop500

res24_pop50  = []
res24_pop100 = []
res24_pop500 = []

res24_fit_pop50  = np.zeros((20, 2001))
res24_fit_pop100 = np.zeros((20, 2001))
res24_fit_pop500 = np.zeros((20, 2001))

for i in range(20):
    plan_ga24_pop50, best_fit_individuals24_pop50 = genetic_algorithm(sub24,␣
 ↪population_size=50)
    plan_ga24_pop100, best_fit_individuals24_pop100 = genetic_algorithm(sub24,␣
 ↪population_size=100)
    plan_ga24_pop500, best_fit_individuals24_pop500 = genetic_algorithm(sub24,␣
 ↪population_size=500)

    res24_pop50.append(plan_ga24_pop50)
    res24_pop100.append(plan_ga24_pop100)
    res24_pop500.append(plan_ga24_pop500)

    res24_fit_pop50[i]  = best_fit_individuals24_pop50
    res24_fit_pop100[i] = best_fit_individuals24_pop100
    res24_fit_pop500[i] = best_fit_individuals24_pop500
```

[ ]:
```
import seaborn as sns
import pandas as pd

data1 = [x[1] for x in res10_pop50]
data2 = [x[1] for x in res10_pop100]
data3 = [x[1] for x in res10_pop500]

fig = plt.figure()
plt.ylabel('Total Path Length')
plt.xlabel('Population Size')
```

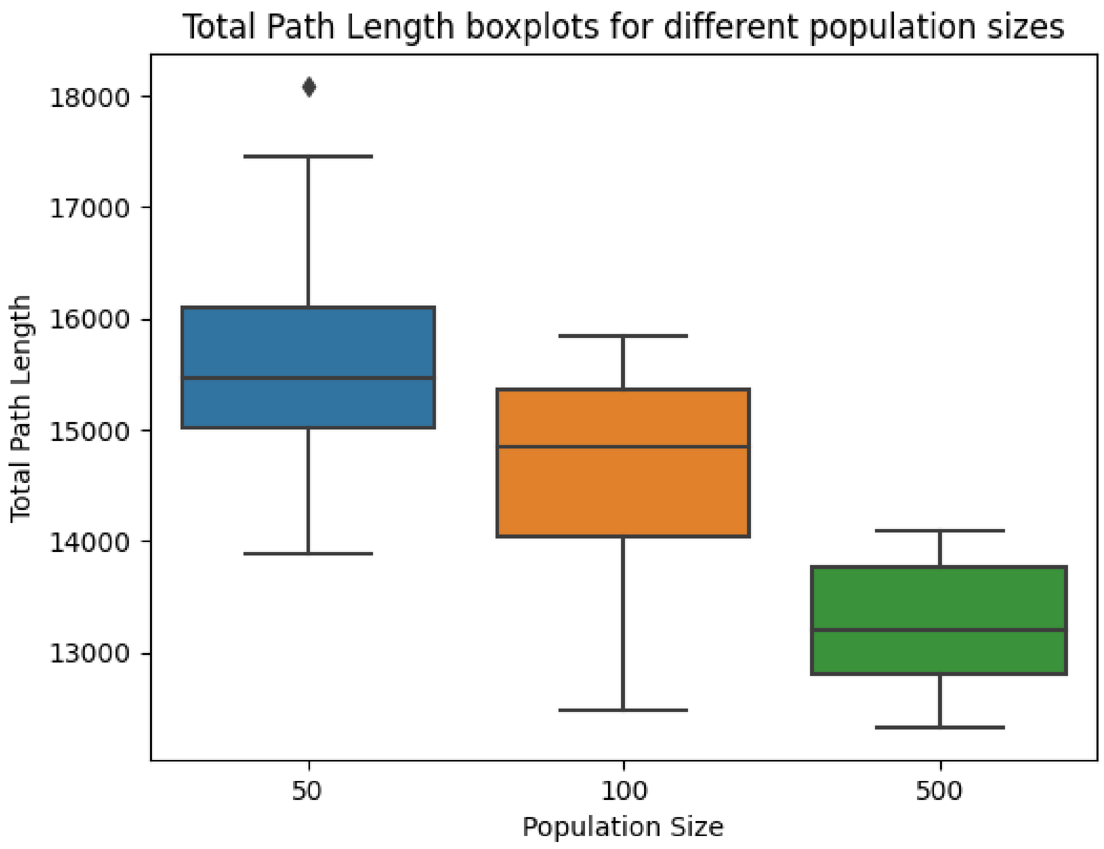
```python
plt.title('Total Path Length boxplots for different population sizes')

df = pd.DataFrame({'50': data1, '100': data2, '500': data3})
sns.boxplot(data=df)

plt.show()

mean1 = np.mean(res10_fit_pop50, axis=0)
mean2 = np.mean(res10_fit_pop100, axis=0)
mean3 = np.mean(res10_fit_pop500, axis=0)

std1 = np.std(res10_fit_pop50, axis=0)
std2 = np.std(res10_fit_pop100, axis=0)
std3 = np.std(res10_fit_pop500, axis=0)

fig = plt.figure()

plt.xlabel('Iterations')
plt.ylabel('Fitness of best invidual (lower is better)')
plt.title('Fitness of best individual over time for different population sizes')

plt.plot(mean1, label='50')
plt.fill_between(range(len(mean1)), mean1 - std1, mean1 + std1, alpha=0.2)

plt.plot(mean2, label='100')
plt.fill_between(range(len(mean2)), mean2 - std2, mean2 + std2, alpha=0.2)

plt.plot(mean3, label='500')
plt.fill_between(range(len(mean3)), mean3 - std3, mean3 + std3, alpha=0.2)

plt.legend()
plt.show()
```

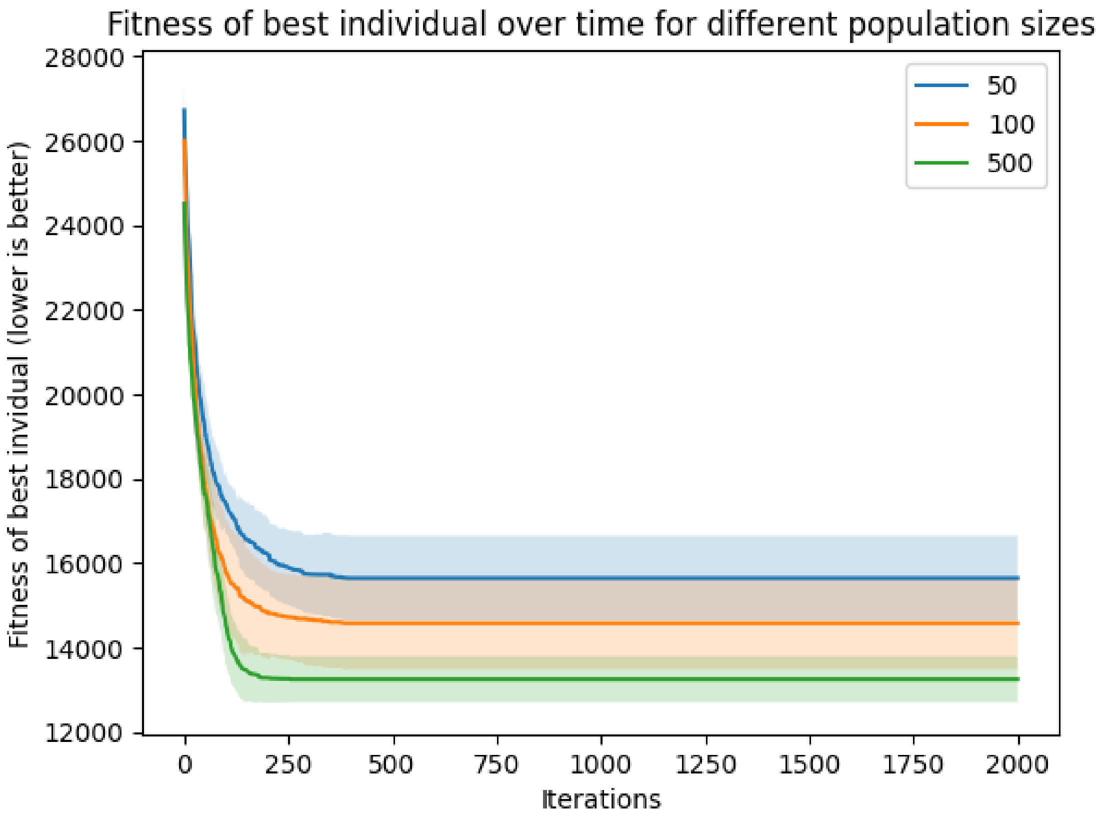

Total Path Length boxplots for different population sizes
Total Path Length
7800
7700
7600
7500
7400
50
100
500
Population Size



```
[ ]: data1 = [x[1] for x in res24_pop50]
     data2 = [x[1] for x in res24_pop100]
     data3 = [x[1] for x in res24_pop500]

     df = pd.DataFrame({'50': data1, '100': data2, '500': data3})

     fig = plt.figure()
     plt.title('Total Path Length boxplots for different population sizes')
     plt.ylabel('Total Path Length')
     plt.xlabel('Population Size')

     sns.boxplot(data=df)

     plt.show()

     mean1 = np.mean(res24_fit_pop50, axis=0)
     mean2 = np.mean(res24_fit_pop100, axis=0)
     mean3 = np.mean(res24_fit_pop500, axis=0)

     std1 = np.std(res24_fit_pop100, axis=0)
     std2 = np.std(res24_fit_pop50, axis=0)
```

```
std3 = np.std(res24_fit_pop500, axis=0)

fig = plt.figure()

plt.xlabel('Iterations')
plt.ylabel('Fitness of best invidual (lower is better)')
plt.title('Fitness of best individual over time for different population sizes')

plt.plot(mean1, label='50')
plt.fill_between(range(len(mean1)), mean1 - std1, mean1 + std1, alpha=0.2)

plt.plot(mean2, label='100')
plt.fill_between(range(len(mean2)), mean2 - std2, mean2 + std2, alpha=0.2)

plt.plot(mean3, label='500')
plt.fill_between(range(len(mean3)), mean3 - std3, mean3 + std3, alpha=0.2)

plt.legend()
plt.show()
```





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?

1.9 Answer

For the first 10 cities our evolutionaayr algorithm seems to always find the best solution. We can observe that increasing the population size increases significantly the accuracy of the final solution, given 2000 iterations, with 24 cities we get close to the optimal solution, we can observe that with a bigger population the risk of getting stuck in a local minimum is reduced.

Compared to the exhaustive search we're doubling the population at every iteration so it's going to be a linear increase in time, based on how big of a starting population we're choosing we're talking about millions, at most, of paths analyzed for the 24 cities path in contrast to exhaustive search which increases factorially, we would need to analyze $24! \approx 6 * 10^{23}$ paths to find the best one.

1.10 Hybrid Algorithm (IN4050 only)

1.10.1 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)

1.10.2 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?

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