IN3050/IN4050 Mandatory Assignment 1: Traveling Salesman Problem (mafredri)

How to run

Spam Shift + Enter

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 28 2025, 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?

Deliver 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, with example runs.
  - 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:

- Include example runs of your code by doing the reports described in the tasks. Simply implementing the code, but never running it will not give many points.
- Include the code that was used to make all reports. Do not include reports of performance and time without also including the code that was used to produce it.
- 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.

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 have to use Python to solve the assignment. 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. The **itertools** module in Python 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 [2]:
```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
np.random.seed(57)
#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], "Berl
    "Brussels": [4.35, 50.85], "Bucharest": [26.10, 44.44], "Budapest": [
    "Copenhagen": [12.57, 55.68], "Dublin": [-6.27, 53.35], "Hamburg": [9
    "Istanbul": [28.98, 41.02], "Kyiv": [30.52, 50.45], "London": [-0.12,
    "Madrid": [-3.70, 40.42], "Milan": [9.19, 45.46], "Moscow": [37.62, 5
    "Munich": [11.58, 48.14], "Paris": [2.35, 48.86], "Prague": [14.42, 5
    "Rome": [12.50, 41.90], "Saint Petersburg": [30.31, 59.94], "Sofia":
    "Stockholm": [18.06, 60.33], "Vienna": [16.36, 48.21], "Warsaw": [21.
```

In [3]:
```
#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],

# 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 [4]:
```
#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

    # 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 [5]:
```
#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', 'Kyiv', 'London', 'Madrid',
'Milan', 'Moscow', 'Munich', 'Paris', 'Prague', 'Rome', 'Saint Petersburg'
, 'Sofia', 'Stockholm', 'Vienna', 'Warsaw']
```

Exhaustive Search

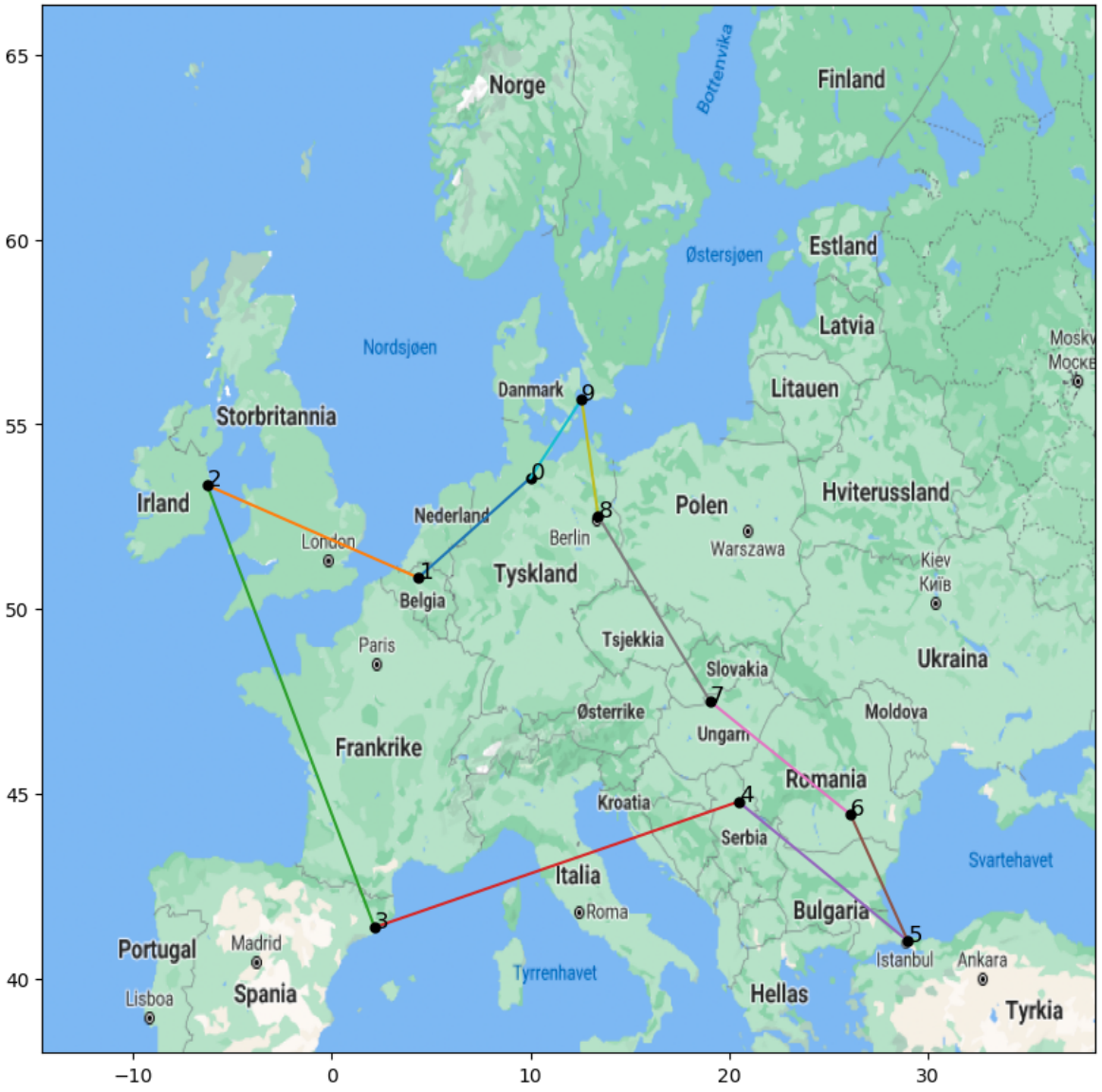
First, try to solve the problem by inspecting every possible tour. Start by writing a program to find the shortest tour among a subset of the cities (say, **6** of them). Measure the amount of time your program takes. Incrementally add more cities and observe how the time increases. Plot the shortest tours you found using the plot_plan method above, for 6 and 10 cities.

Note: To get distances between cities, use the dictionary `data` created by reading the file `european_cities.csv`. *Do not* calculate distances based on the coordinates. The actual distances do not only depend on the differences in the coordinates, but also of the curvature of the earth. The distances available in `data` are corrected for this, and contain the actual true distances.

In [12]:
```
# Implement the algorithm here
from itertools import permutations
from time import time

def e_search(n: int, data: list):
    perms = permutations(range(n))
    shortest_distance = float("inf")
    shortest_path = next(iter(perms))

    for p in perms:
        distance = get_total_distance(p, data)

        if distance < shortest_distance:
            shortest_distance = distance
            shortest_path = p

    return shortest_distance, shortest_path

def get_total_distance(cities: tuple, data: list):
    distance = 0

    for i in range(-1, len(cities)-1):
        current_city = cities[i]
        next_city = cities[i+1]
        distance += float(data[current_city][next_city])

    return distance

n = 10
start = time()
distance, seq = e_search(n, data[1:])
end = time()
print("Distance: ", distance)
print("Sequence: ")
for s in seq:
    print(data[0][s])
time = end-start
print(f"Time taken ({n} cities): ", time)
print("Time taken (24 cities): ", sum([time := time * a for a in range(n+

best_plan = [data[0][c] for c in seq]
```

```
plot_plan(best_plan)
```

```
Distance:  7486.309999999999
Sequence:
Hamburg
Brussels
Dublin
Barcelona
Belgrade
Istanbul
Bucharest
Budapest
Berlin
Copenhagen
Time taken (10 cities):  5.765214443206787
Time taken (24 cities):  1.0286733711684381e+18
```



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?

Answer

Shortest distance for the first 10 cities:

7486.309km

Sequence of cities in shortest path:

Hamburg
Brussels
Dublin
Barcelona
Belgrade
Istanbul
Bucharest
Budapest
Berlin
Copenhagen

Estimated time for with 24 cities:

By finding the time it takes for 10 cities it takes 3 seconds. Since this is a brute force algorithm with O(n!) run time it will take 3 * 11 seconds for 11 cities. By continuing this way of calculating time it will take the algorithm 1.8613452e+24 seconds to finish with 24 cities as input.

What I have done

In the algorithm I check for each and single combination and used the best one. This is slow.

Hill Climbing

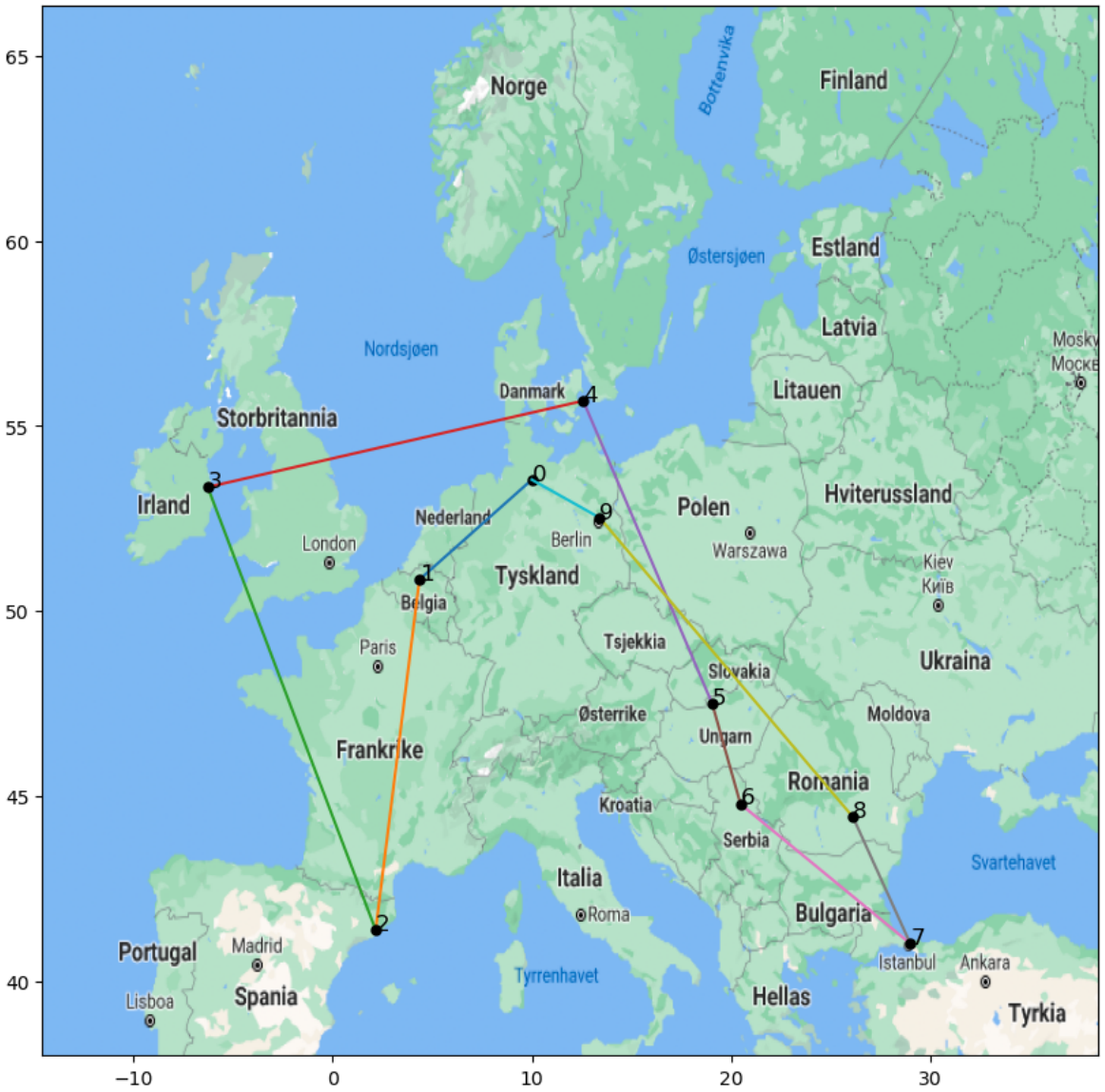
Then, write a simple hill climber to solve the TSP. How well does the hill climber perform, compared to the result from the exhaustive search for the first **10 cities**? Since you are dealing with a stochastic algorithm, you should run the algorithm several times to measure its performance. Report the 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 [15]:
```
# Implement the algorithm here
from random import shuffle
from time import time

def h_search(cities: list[int], data: list):
    n = len(cities)
    shortest_path = cities
```

```
    shortest_distance = get_total_distance(cities, data)

    while True:
        cities_list = get_neighbours(shortest_path)
        next_path = []

        next_shortest_distance = shortest_distance
        next_shortest_path = shortest_path

        for neighbour in cities_list:
            current_distance = get_total_distance(neighbour, data)
            if current_distance < next_shortest_distance:
                next_shortest_distance = current_distance
                next_shortest_path = neighbour

        if next_shortest_distance < shortest_distance:
            shortest_distance = next_shortest_distance
            shortest_path = next_shortest_path
            continue

        break

    return shortest_distance, shortest_path

def get_neighbours(cities: list[int]):
    n = len(cities)
    cities_list = []

    for i in range(n):
        for j in range(n):
            if i != j:
                nc = cities.copy()
                nc[i], nc[j] = nc[j], nc[i]
                cities_list.append(nc)

    return cities_list



def run_test(n: int, k: int):
    start = time()
    d_sum = 0
    d_min = float('inf')
    d_max = float('-inf')

    p_min = []
    p_max = []

    d_data = []

    for _ in range(k):
        cities = list(range(n))
        shuffle(cities)

        d, p = h_search(cities, data[1:])
        d_data.append(d)
        d_sum += d
        if d < d_min:
            d_min = d
            p_min = p
```

```
        if d > d_max:
            d_max = d
            p_max = p
    end = time()

    print("Distance (avg): ", (d_sum / k))
    print("Distance (min): ", d_min)
    print("Distance (max): ", d_max)
    print("Distance (std): ", np.std(d_data))
    print("Time used: ", (end-start))

    plan_min = [data[0][c] for c in p_min]
    plan_max = [data[0][c] for c in p_max]

    plot_plan(plan_min)
    plot_plan(plan_max)

run_test(10, 20)
run_test(24, 20)
```

```
Distance (avg):  7585.233
Distance (min):  7486.3099999999995
Distance (max):  8391.05
Distance (std):  209.43467284812215
Time used:  0.019824981689453125
```




```
Distance (avg):  14153.176000000003
Distance (min):  12520.17
Distance (max):  15753.160000000002
Distance (std):  977.4056000985465
Time used:  0.6521091461181641
```


Norge
Finland
Bottenvika
Østersjøen
Estland
Latvia
Litauen
Nordsjøen
Danmark
Storbritannia
Irland
Nederland
Polen
Hviterussland
Berlin
Warszawa
London
Tyskland
Belgia
Kiev
Tsjekkia
Slovakia
Ukraina
Østerrike
Moldova
Ungarn
Frankrike
Romania
Kroatia
Serbia
Svartehavet
Italia
Roma
Bulgaria
Portugal
Madrid
Spania
Lisboa
Tyrrenhavet
Istanbul
Ankara
Hellas
Tyrkia
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
65
60
55
50
45
40
−10
0
10
20
30

Answer

Time comparrison of Exhaustive Search

The hillclimber algorithm is faster than the exhaustive search algorithm, but the answer may not be correct with hillclimb.

Results

The results are in the prints after running the program.

Thought process

I generate a random instance of the problem. After that I find each neighbour of the instance and find the best one. I loop this process until there is no better neighbour. Creating the neighbours consist of combinations of swapped cities, but not all permetations or else it would be the same as exhaustive search.

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. This means that the x-axis should be the generations over time and the y-axis should be the average (over the 20-runs) fitness of the best gene in that generation. 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 [25]:
```
# Implement the algorithm here
from random import shuffle
from random import randint
from random import choice
from random import random
from time import time


def pmx_pair(seq1, seq2):
    n = min(len(seq1), len(seq2))
    c1, c2 = [0] * n, [0] * n


    p1 = randint(0, n-2)
    p2 = randint(p1+1,n)

    cd1 = seq1[p1:p2]
    cd2 = seq2[p1:p2]

    x_map = {k:v for k, v in zip(cd1, cd2)}
    x_map_r = {v:k for k, v in x_map.items()}

    for i in range(n):
        if i >= p1 and i < p2:
            c1[i] = seq2[i]
            c2[i] = seq1[i]
        else:
            current = seq1[i]
            while current in x_map_r:
                current = x_map_r[current]
            c1[i] = current

            current = seq2[i]
            while current in x_map:
```

```
                current = x_map[current]
            c2[i] = current

    return c1, c2

def mutate_offsprings(offsprings: list[list[int]], m_rate: float, c_degre
    for offspring in offsprings:
        n = len(offspring)-1
        if random() < m_rate:
            for _ in range(c_degree):
                i = randint(0, n)
                j = randint(0, n)
                while j == i:
                    j = randint(0, n)
                offspring[i], offspring[j] = offspring[j], offspring[i]


def f_func(individual: list[int], data: list) -> float:
    return get_total_distance(individual, data)

def p_select(population: list[list[int]], p_size: int, n_elites: int, dat
    parents = population[:n_elites]
    rest = population[n_elites:]
    population_distance = sum([f_func(p, data) for p in rest])

    select_chances = [f_func(p, data)/population_distance for p in rest][

    n_offsprings = p_size // 2
    n = len(rest)
    i = 0

    while len(parents) < n_offsprings:
        if random() < select_chances[i]:
            parents.append(rest[i])
        i = (i + 1) % 1

    return parents



def create_offsprings(parents: list[list[int]]) -> list[list[int]]:
    offsprings = []
    n = len(parents)

    for i in range(0, n, 2):
        c1, c2 = pmx_pair(parents[i], parents[i+1])
        offsprings.append(c1)
        offsprings.append(c2)

    return offsprings



def s_selection(offsprings: list[list[int]], population: list[list[int]],
    new_population = list(sorted(offsprings + population, key=lambda x: f
    return new_population


def g_search(c_size: int, p_size: int, generations: int, m_rate: float, c
    population = []
```

```
    # initilize the population
    for _ in range(p_size):
        individual = list(range(c_size))
        shuffle(individual)
        population.append(individual)
    population.sort(key=lambda x: f_func(x, data))

    for _ in range(generations):
        parents = p_select(population, p_size, n_elites, data)
        offsprings = create_offsprings(parents)
        mutate_offsprings(offsprings, m_rate, c_degree)
        population = s_selection(offsprings, population, data)

    return population

c_size = 24
p_size = 100
generations = 100
m_rate = 0.9
c_degree = 1
n_elites = 5

for pop in [100, 500, 1000]:
    for c_s in [10, 24]:
        start = time()
        f_data = []
        min_path = []
        max_path = []
        shortest = float('inf')
        longest = float('-inf')

        r = 20
        for _ in range(r):
            population = g_search(c_s, pop, generations, m_rate, c_degree

            p = population[0]
            f = f_func(p, data[1:])
            f_data.append(f)

            if f < shortest:
                shortest = f
                min_path = p
            if f > longest:
                longest = f
                max_path = p
        end = time()
        print(f"""
        Amount of cities:   {c_s}
        Population size:    {pop}
        Generations:        {generations}
        Mutation rate:      {m_rate}
        Change degree:      {c_degree}
        Number of elites:   {n_elites}

        Average Fitness:    {sum(f_data) / r:.2f}km
        Best Fitness:       {min(f_data):.2f}km
        Worst Fitness:      {max(f_data):.2f}km
        Standard Deviation: {np.std(f_data):.2f}km
        Time used:          {end-start:.2f}s
```

```
        ----------------------------------------
        """)

        # plan_min = [data[0][c] for c in min_path]
        # plan_max = [data[0][c] for c in max_path]

        # plot_plan(plan_min)
        # plot_plan(plan_max)
```

```
Amount of cities:   10
Population size:    100
Generations:        100
Mutation rate:      0.9
Change degree:      1
Number of elites:   5

Average Fitness:    7500.57km
Best Fitness:       7486.31km
Worst Fitness:      7737.95km
Standard Deviation: 54.69km
Time used:          2.24s
----------------------------------------


Amount of cities:   24
Population size:    100
Generations:        100
Mutation rate:      0.9
Change degree:      1
Number of elites:   5

Average Fitness:    13760.07km
Best Fitness:       12740.64km
Worst Fitness:      14877.91km
Standard Deviation: 671.29km
Time used:          3.53s
----------------------------------------


Amount of cities:   10
Population size:    500
Generations:        100
Mutation rate:      0.9
Change degree:      1
Number of elites:   5

Average Fitness:    7508.94km
Best Fitness:       7486.31km
Worst Fitness:      7737.95km
Standard Deviation: 58.38km
Time used:          7.81s
----------------------------------------


Amount of cities:   24
Population size:    500
Generations:        100
Mutation rate:      0.9
Change degree:      1
Number of elites:   5

Average Fitness:    13924.78km
Best Fitness:       12514.02km
Worst Fitness:      15776.49km
Standard Deviation: 828.56km
Time used:          10.17s
----------------------------------------
```

```
Amount of cities:   10
Population size:    1000
Generations:        100
Mutation rate:      0.9
Change degree:      1
Number of elites:   5

Average Fitness:    7569.76km
Best Fitness:       7486.31km
Worst Fitness:      8349.94km
Standard Deviation: 199.68km
Time used:          15.46s
----------------------------------------


Amount of cities:   24
Population size:    1000
Generations:        100
Mutation rate:      0.9
Change degree:      1
Number of elites:   5

Average Fitness:    13898.79km
Best Fitness:       12325.93km
Worst Fitness:      16419.94km
Standard Deviation: 886.03km
Time used:          19.28s
----------------------------------------
```

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?

Answer

First 10 cities comparison

For the first 10 cities the GA algorithm sometimes find the best solution but not always. Other than that its very close most of the times.

Running time for 10 and 24 cities, GA vs Exhaustive Search

The run time for GA is the same or shorter as the Exhaustive Search with 3 seconds.

Tours inspected

The GA will visit 100 cities to start with and then create 50 offsprings (product of

offsprings and the offsprings being potentially mutated) for each iteration. Based on the arguments above we get 100 + 50 * 100.

Im not sure if this is 100% though, but the number 50 (from offsprings) isnt truly 50. It should be more based on the mutation rate since the crossovers will be more alike. Instead of 50 it should be (50 * mutation rate) which is in this case 0.9.

Results

The results are in pngs in this folder. The results shows that increasing the population size doesnt increase the accuracy significantly, but the runtime increases.

Thought process

When creating this algorithm I followed the steps of a typical EA. First create a population of random instances of the problem. After that run the following n_generation times:

- Pick out some of the population, parents
  - Her I use a elite strategy and roulette one. Always keep the 5 best for example and pick out the rest by chance.
- Use the parents to create offsprings with pmx and mutate the offsprings with a set mutation rate
  - I used pmx since there was a need to just randomize the order. With pmx we can do that since it just chops the genotypes in chunks and sets it together kind of.
  - For the mutation I just swapped two random cities (the number depends on the change_degree) placements.
- Use the best individuals to overwrite what the last population was to be a combination of the offsprings and the population, but keep the population size.

The fitness function is just the distance of the tour.

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 3)

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
```