IN3050/IN4050 Mandatory Assignment 1: Traveling Salesman Problem

by Florian Arbes (floriaa)

Introduction

In this exercise, an instance of the traveling salesman problem (TSP) is solved using different methods. The doc folder contains a .csv file with 24 cities and their distance to each other.

	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.

A traveling salesman wants to visit the residents of the major cities in some region of the world in the shortest time possible. He 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 file european_cities.csv.

In a first step, a map was created in order to visualize the problem (see: make_a_nice_map.py). This is very helpful for debugging as visualizations are a quick way to see what is happening.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

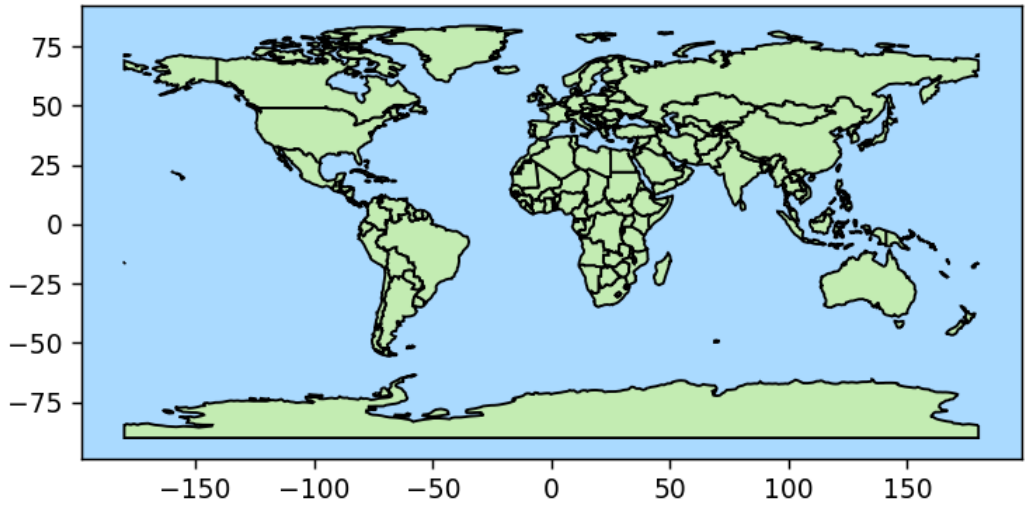
The map is saved as a pickle file, so it will be quick and easy accessible. Don't worry if this doesn't run for you, the files are saved in the doc folder. The exact location of the 24 bespoken cities are taken from a file (source: https://simplemaps.com/data/world-cities) and saved as well for the same reasons.

In [1]:
```
# -*- coding: utf-8 -*-
"""
Created on Sun Feb  9 12:36:34 2020

@author: Florian
sources:
    https://stackoverflow.com/a/50907364/6747238
    https://stackoverflow.com/a/48056459/6747238
    https://simplemaps.com/data/world-cities
    https://www.schemecolor.com/google-map-basic-colors.php
"""
from pprint import pprint
import matplotlib.pyplot as plt
import cartopy.io.shapereader as shpreader
import pickle
from descartes import PolygonPatch
import numpy as np
%matplotlib notebook


shapename = 'admin_0_countries'
countries_shp = shpreader.natural_earth(resolution='110m',
                                        category='cultural', name=shape
name)
fig, ax = plt.subplots()
ax.set_facecolor('#aadaff')

for country in shpreader.Reader(countries_shp).records():
#    print(country.attributes['NAME_LONG'])
    poly = country.geometry.__geo_interface__
    ax.add_patch(PolygonPatch(poly, fc='#c3ecb2', ec='#000000',
                              alpha=1.0, zorder=2))
ax.axis('scaled')
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
plt.show()

with open('../doc/world_map.pkl', 'wb') as fid:
    pickle.dump(ax, fid)

#  make file with latitude and longitude
city_names = ["Barcelona", "Belgrade", "Berlin", "Brussels", "Buchares
t",
              "Budapest", "Copenhagen", "Dublin", "Hamburg", "Istanbul"
,
              "Kyiv", "London", "Madrid", "Milan", "Moscow", "Munich",
              "Paris", "Prague", "Rome", "Saint Petersburg", "Sofia",
              "Stockholm", "Vienna", "Warsaw"]
latlon = np.empty((len(city_names), 2))
latlon_txt = np.genfromtxt("../doc/worldcities.txt", dtype="<U256",
                           delimiter="\t", skip_header=1)
for i, cn in enumerate(city_names):
    ll = latlon_txt[latlon_txt[:, 0] == cn][0]
    latlon[i] = [np.float64(ll[1]), np.float64(ll[2])]
np.savetxt("../doc/latitude_lonitude.txt", latlon, delimiter=";",
           header=';'.join(city_names))
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD



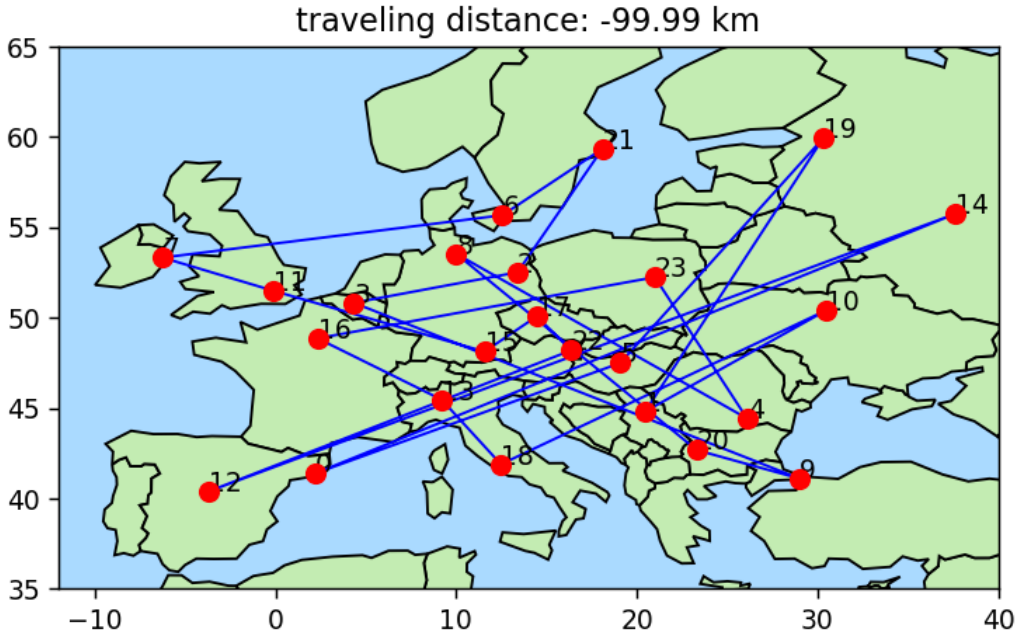
All of the Tasks are implemented in Assignment1.py. The main method runs test functions, that correspond to one task each. The same code can be found in this jupyter notebook. The first functions is the bespoken plotting function, that displays the path of the traveling salesman on a nice map:

In [2]:
```
import pickle
import matplotlib.pyplot as plt
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
def plot_solution(dist, order, latlon):
    %matplotlib notebook
    with open('../doc/world_map.pkl', 'rb') as fid:
        ax_ = pickle.load(fid)
    xmin, xmax = -12, 40
    ymin, ymax = 35, 65
    plt.xlim(xmin, xmax)
    plt.ylim(ymin, ymax)
    for i in range(len(latlon)):
        plt.text(latlon[i, 1], latlon[i, 0], str(i), fontsize=10)
    ax_.fill(latlon[order, 1], latlon[order, 0], edgecolor='b',
             fill=False, zorder=3)
    ax_.scatter(latlon[order, 1], latlon[order, 0],
                s=50, c="r", zorder=4)
    plt.title("traveling distance: {:.2f} km".format(dist))
    plt.show()
    return


latlon = np.genfromtxt("../doc/latitude_lonitude.txt", delimiter=";", s
kip_header=1)
plot_solution(-99.99, np.random.permutation(24), latlon)
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD



Q1) 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.

First of all, a objective function has to be defined. It takes the *city order* of the travel plan and the distance between the cities as argument and returns the distance of that travel plan. Also one

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

might want to know haw many different travel plans there are in theory. This can be calculated with the factorial function.

In [3]:
```
def travel_distance(distances, order):
    d = 0.0
    for i in range(len(order)):
        c1 = order[i-1]
        c2 = order[i]
        d += distances[c1, c2]
    return d


def factorial(n):
    fact = 1
    for num in range(2, n + 1):
        fact *= num
    return fact
```

The exhaustive search itself is pretty straight forward: one has to iterate over all permutations of *city orders* there are, while keeping track of the shortest distance:

In [4]:
```
import numpy as np
import time
from itertools import permutations


def exhaustive_search(distances):
    N = len(distances)
    opt_dist = np.inf
    opt_order = None
    for i, order in enumerate(permutations(np.arange(N))):
        d = travel_distance(distances, order)
        if d < opt_dist:
            opt_dist = d
            opt_order = order
    return opt_order, opt_dist
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
distances = np.genfromtxt("../doc/european_cities.csv",
                          delimiter=";", skip_header=1)
print("Exhaustive search:\n--------------------------")
for N in [6, 7, 8, 9, 10]:
    print(N, "cities ({:.0f} options)".format(factorial(N)))
    tic = time.perf_counter()
    opt_order, opt_dist = exhaustive_search(distances[:N, :N])
    toc = time.perf_counter()
    print("run time: {:.3f} seconds".format(toc - tic))
    print("shortest tour: {:.2f} km".format(opt_dist))
    print("city order: ", opt_order, "\n")
```

```
Exhaustive search:
--------------------------
6 cities (720 options)
run time: 0.003 seconds
shortest tour: 5018.81 km
city order:  (1, 0, 3, 2, 5, 4)

7 cities (5040 options)
run time: 0.032 seconds
shortest tour: 5487.89 km
city order:  (2, 6, 3, 0, 1, 4, 5)

8 cities (40320 options)
run time: 0.255 seconds
shortest tour: 6667.49 km
city order:  (3, 7, 0, 1, 4, 5, 2, 6)

9 cities (362880 options)
run time: 2.125 seconds
shortest tour: 6678.55 km
city order:  (3, 7, 0, 1, 4, 5, 2, 6, 8)

10 cities (3628800 options)
run time: 22.640 seconds
shortest tour: 7486.31 km
city order:  (8, 3, 7, 0, 1, 9, 4, 5, 2, 6)
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

This inspections shows, the number of possible tours explodes quickly and with it the runtime increases.

Q1.1) 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)?

shortest tour: 7486.31 km

city order: (8, 3, 7, 0, 1, 9, 4, 5, 2, 6)

see map below.

Q1.2) How long did your program take to find it?

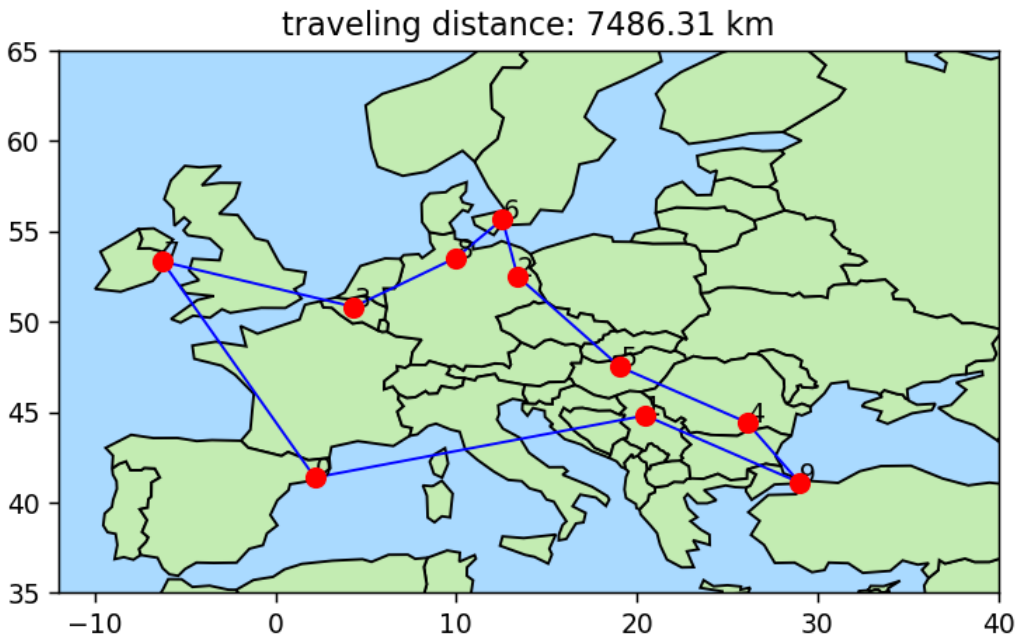
about 23 seconds.

Q1.3) Calculate an approximation of how long it would take to perform exhaustive search on all 24 cities?

In [5]:
```
print("computation time for an exhaustive seach of the shortest tour be
tween 24 cities: {:.0f} years".format(23/60/60/24/365/factorial(10)*fac
torial(24)))
plot_solution(opt_dist, opt_order, latlon[:N, :N])
```

```
computation time for an exhaustive seach of the shortest tour between 2
4 cities: 124699257159 years
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD



Q2) Hill Climbing

Then, write a simple hill climber to solve the TSP.

This algorithm starts with a random permutation and then tests if swapping **any** two cities in the travel plan improves the traveling distance. This can be seen as a small local change of the input parameters. The algorithm stops when the route can no longer be improved.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

In [6]:
```
def hillclimb_search(distances, order=False, maxiter=np.inf):
    N = len(distances)
    if order is False:
        order = np.random.permutation(N)
    opt_dist = np.inf
    route_improved = True
    count = 0
    while route_improved is True and count < maxiter:
        count += 1
        route_improved = False
        for i in range(N):
            for j in range(i):
                if np.sum(order) != np.sum(np.arange(N)):
                    print(count, i, j, order)
                tmp = order[j]
                order[j] = order[i]
                order[i] = tmp
                d = travel_distance(distances, order)
                if d < opt_dist:
                    opt_dist = d
                    opt_order = order.copy()
                    route_improved = True
                else:  # change back rather than copying
                    tmp = order[j]
                    order[j] = order[i]
                    order[i] = tmp
    return opt_order, opt_dist
```

Q2.1) How well does the hill climber perform, compared to the result from the exhaustive search for the first 10 cities?

In [7]:
```
tic = time.perf_counter()
opt_order, opt_dist = hillclimb_search(distances[:10, :10])
toc = time.perf_counter()
dt = toc-tic
print("runtime: {:.4f} seconds. (= {:.2f} % compared to the exhaustive
 search)".format(dt, dt/23*100))
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
runtime: 0.0067 seconds. (= 0.03 % compared to the exhaustive search)
```

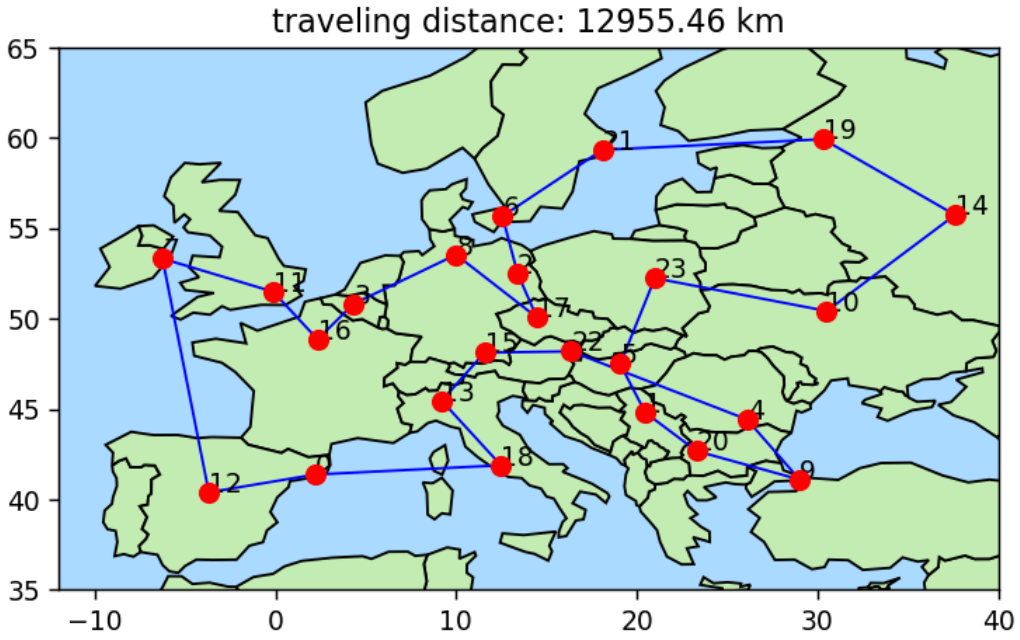
It only needs a very small fraction of the time, as it only evaluates a small fraction of all possible travel paths.

Q2.2) 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 [8]:
```
tour_length = np.empty(20, )
for n_cities in [10, 24]:
    print("number of cities:", n_cities, end="")
    for i in range(20):
        opt_order, opt_dist = hillclimb_search(distances[:n_cities, :n_
cities])
        tour_length[i] = opt_dist
    print("""
    shortest tour: {:.2f} km
    longest tour: {:.2f} km
    average tour: {:.2f} km
    standard deviation {:.2f} km """.format(np.min(tour_length), np.max
(tour_length),
                                            np.mean(tour_length), np.st
d(tour_length)))
plot_solution(opt_dist, opt_order, latlon)
```

```
number of cities: 10
    shortest tour: 7486.31 km
    longest tour: 8407.18 km
    average tour: 7601.14 km
    standard deviation 212.47 km
number of cities: 24
    shortest tour: 12955.46 km
    longest tour: 16082.25 km
    average tour: 14712.68 km
    standard deviation 859.48 km
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD



As the algorithm starts with a random permutation, it might converge to a local optimum, which may or may not be the best solution. Running the algorithm 20 times lead to the shortest travel path between 10 cities at least once in this case. The shortest travel path between 24 cities does not seem to be optimal (visually).

Q3) Genetic Algorithm

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Q3.1) write a genetic algorithm (GA) to solve the problem.

The genetic algorithm itself is straight foreward and follows the steps from the textbook very closely. Parent selection, survivor selection, and recombination operators may be arbitrary ("dependency injection").

In [9]:
```
def evolutution_TSP(distances, n_survivors, n_parents, n_generations,
                    parent_selection, recombination, survivor_selection
,
                    hybrid=None):
    max_dist = np.sum(np.max(distances, axis=1))
    n_children = n_parents
    n_alleles = len(distances)
    children = np.empty((n_children, n_alleles), dtype=np.int32)
    survivors = np.empty((n_survivors, n_alleles), dtype=np.int32)
    population = np.empty((n_survivors+n_children, n_alleles), dtype=np
.int32)
    fitness_chrn = np.empty((n_children,))
    fitness_surv = np.empty((n_survivors,))
    fitness_popu = np.empty((n_survivors+n_children,))
    shortest_dist = np.empty(n_generations,)
    # INITIALISE population with random candidate solution
    for i in range(n_survivors):
        survivors[i] = np.random.permutation(n_alleles)
        fitness_surv[i] = max_dist-travel_distance(distances, survivors
[i])

    for generation in range(n_generations):
        # SELECT parents:
        parent_pairs = parent_selection(fitness_surv, n_parents)
        c = 0
        for parents in parent_pairs:
            gene_p1, gene_p2 = survivors[parents[0]], survivors[parents
[1]]
            # RECOMBINE pairs of parents
            new_genes = recombination(gene_p1, gene_p2)
            # MUTATE the resulting offsprings
            child1, child2 = mutate(new_genes[0]), mutate(new_genes[1])
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
            # EVALUATE new candidates
            if hybrid == "Lamarck":
                # In general the algorithms are referred to as Lamarcki
an if
                # the result of the local search stage replaces the ind
ividual
                # in the population.
                opt_child1, opt_dist1 = hillclimb_search(distances, chi
ld1, 1)
                opt_child2, opt_dist2 = hillclimb_search(distances, chi
ld2, 1)
                f1 = max_dist-travel_distance(distances, opt_child1)
                f2 = max_dist-travel_distance(distances, opt_child2)
                child1, child2 = opt_child1, opt_child2
            elif hybrid == "Baldwinian":
                # In general the algorithms are referred to as Baldwini
an if
                # the original member is kept, but has as its fitness t
he value
                # belonging to the outcome of the local search process.
                opt_child1, opt_dist1 = hillclimb_search(distances, chi
ld1, 1)
                opt_child2, opt_dist2 = hillclimb_search(distances, chi
ld2, 1)
                f1 = max_dist-travel_distance(distances, opt_child1)
                f2 = max_dist-travel_distance(distances, opt_child2)
            else:
                f1 = max_dist-travel_distance(distances, child1)
                f2 = max_dist-travel_distance(distances, child2)

            children[c], children[c+1] = child1, child2
            fitness_chrn[c], fitness_chrn[c+1] = f1, f2
            c += 2
        # SELECT individuals for new generation
        survivor_selection(population, survivors, children,
                           fitness_popu, fitness_surv, fitness_chrn)
        best_candidate = survivors[np.argmax(fitness_surv)]
        shortest_dist[generation] = travel_distance(distances, best_can
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
didate)
    return shortest_dist
```

Q3.2) Choose mutation and crossover operators that are appropriate for the problem (see chapter 4.5 of the Eiben and Smith textbook).

The following mutations may occure with a customizable frequency: **scramble**, **swap**, **insert** or **inversion**. These are implemented in the function `mutate` which can be found in recombinations_and_mutations.py

The survivor selection may be **elitism**, **rank based**, **uniform** or **fitness proportionate**. Adequate functions can be found in selection.py. They can be imported and used in the genetic algorithm ("dependency injection").

The same goes for the recombination operators, you may choose between a **partially mapped crossover**, a **cycle crossover** or an **ordered crossover**.

In [10]:
```
from selection import elitism, rank_based_selection
#from selection import fitness_proportionate_selection, uniform_selecti
on
from recombinations_and_mutations import mutate, \
                    pmx_pair, cycle_crossover_pair, order_crossover_pai
r
```

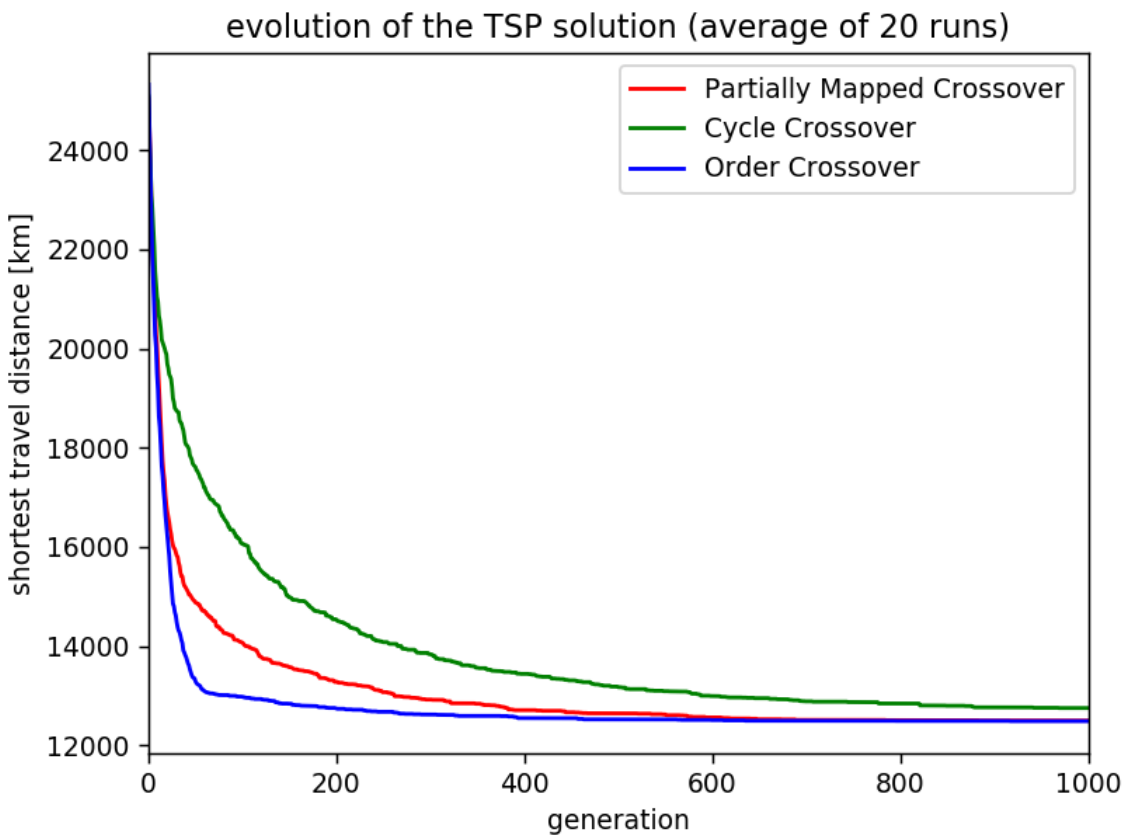
First, we want to find out which recombination operator works best for this problem and should be used for the upcoming investigations:

In [11]:
```
n_generations = 1000
n_runs = 20
fig, ax = plt.subplots()
b = np.empty((3, n_runs, n_generations))
x = np.arange(n_generations)
recombinations = [pmx_pair, cycle_crossover_pair, order_crossover_pair]
names = ["Partially Mapped Crossover", "Cycle Crossover", "Order Crosso
ver"]
colors = ["r", "g", "b"]
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
for i in range(3):
    c = colors[i]
    recombination = recombinations[i]
    print("run: ", end="")
    for j in range(n_runs):
        print(j, end=", ")
        b[i, j] = evolutution_TSP(distances, 100, 100, n_generations,
                                  rank_based_selection, recombination, eli
tism)
    y = np.mean(b[i], axis=0)
    plt.plot(x, y, c+"-", label=names[i])
plt.legend()
plt.xlim(0, n_generations)
plt.xlabel("generation")
plt.ylabel("shortest travel distance [km]")
plt.title("evolution of the TSP solution (average of 20 runs)")
plt.show()
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD



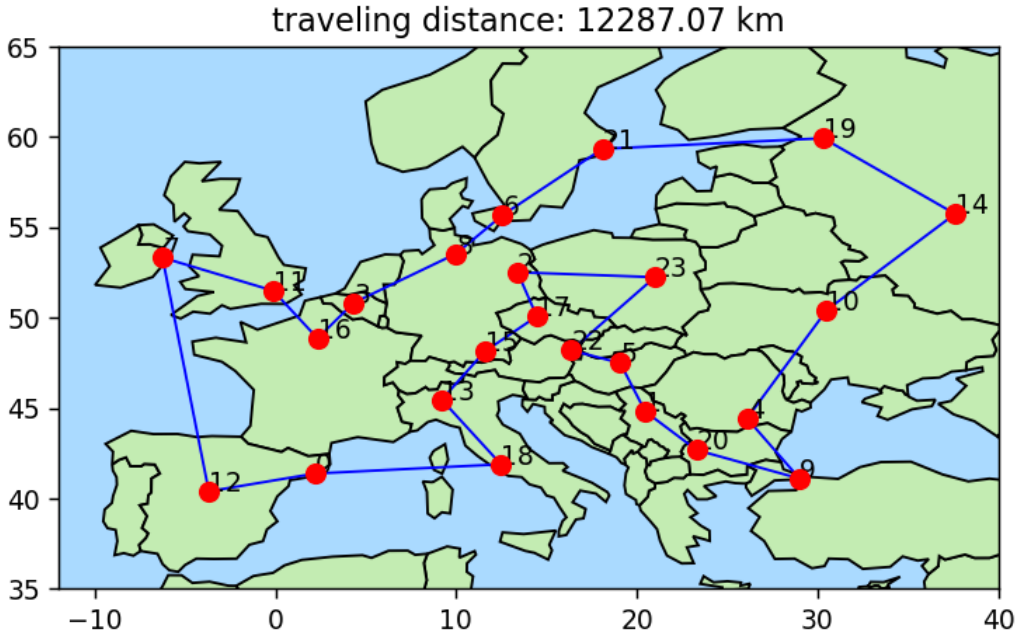

```
run: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, run: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
18, 19, run: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19,
```

The investigation above shows, that the order crossover is best suitable for this problem. Also playing with the mutation frequency showed, a frequency around 0.01 is good. This means a 1 % chance for each possible mutation to occure.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

The following is the shortest tour that was found:

In [12]:
```
best = np.array([17, 2, 23, 22, 5, 1, 20, 9, 4, 10, 14, 19, 21, 6, 8, 3
,
                 16, 11, 7, 12, 0, 18, 13, 15])  # 12287.07
d = travel_distance(distances, best)
plot_solution(d, best, latlon)
```



Q3.3) Choose three different values for the population size. Define and tune other parameters yourself and make assumptions as necessary (and report them, of course).

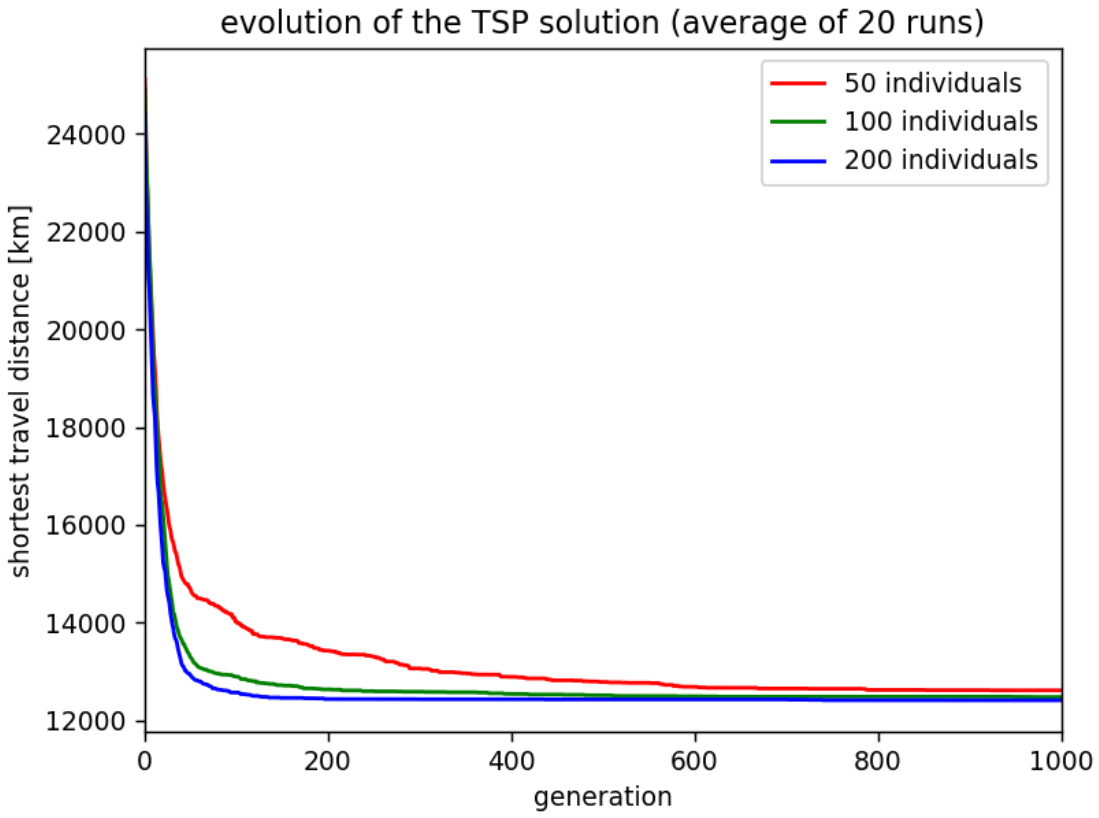
Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

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 [13]:
```
n_generations = 1000
n_runs = 20
tour_length = np.empty(20, )
fig, ax = plt.subplots()
b = np.empty((3, n_runs, n_generations))
x = np.arange(n_generations)
colors = ["r", "g", "b"]
for i, n_survivors in enumerate([50, 100, 200]):
    c = colors[i]
    n_parents = n_survivors
    print("number of individuals:", n_survivors)
    print("run: ", end="")
    for j in range(n_runs):
        print(j, end=", ")
        b[i, j] = evolututution_TSP(distances, n_survivors, n_parents,
                                    n_generations, rank_based_selection,
                                    order_crossover_pair, elitism)
        tour_length[j] = b[i, j, -1]
    print("""
    shortest tour: {:.2f} km
    longest tour: {:.2f} km
    average tour: {:.2f} km
    standard deviation {:.2f} km """.format(np.min(tour_length), np.max
(tour_length),
                                            np.mean(tour_length), np.st
d(tour_length)))
    y = np.mean(b[i], axis=0)
    plt.plot(x, y, c+"-", label="{:.0f} individuals".format(n_survivors
))
plt.legend()
plt.xlim(0, n_generations)
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
plt.xlabel("generation")
plt.ylabel("shortest travel distance [km]")
plt.title("evolution of the TSP solution (average of 20 runs)")
plt.show()
```




```
number of individuals: 50
run: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 1
8, 19,
    shortest tour: 12287.07 km
    longest tour: 13367.90 km
    average tour: 12617.65 km
    standard deviation 307.79 km

number of individuals: 100
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
number of individuals: 100
run: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 1
8, 19,
    shortest tour: 12287.07 km
    longest tour: 13283.27 km
    average tour: 12480.37 km
    standard deviation 248.11 km
number of individuals: 200
run: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 1
8, 19,
    shortest tour: 12287.07 km
    longest tour: 12722.56 km
    average tour: 12412.90 km
    standard deviation 158.62 km
```

It can be concluded that a larger population size leads to better results in earlier generations. The number of evaluation however increases with the population size.

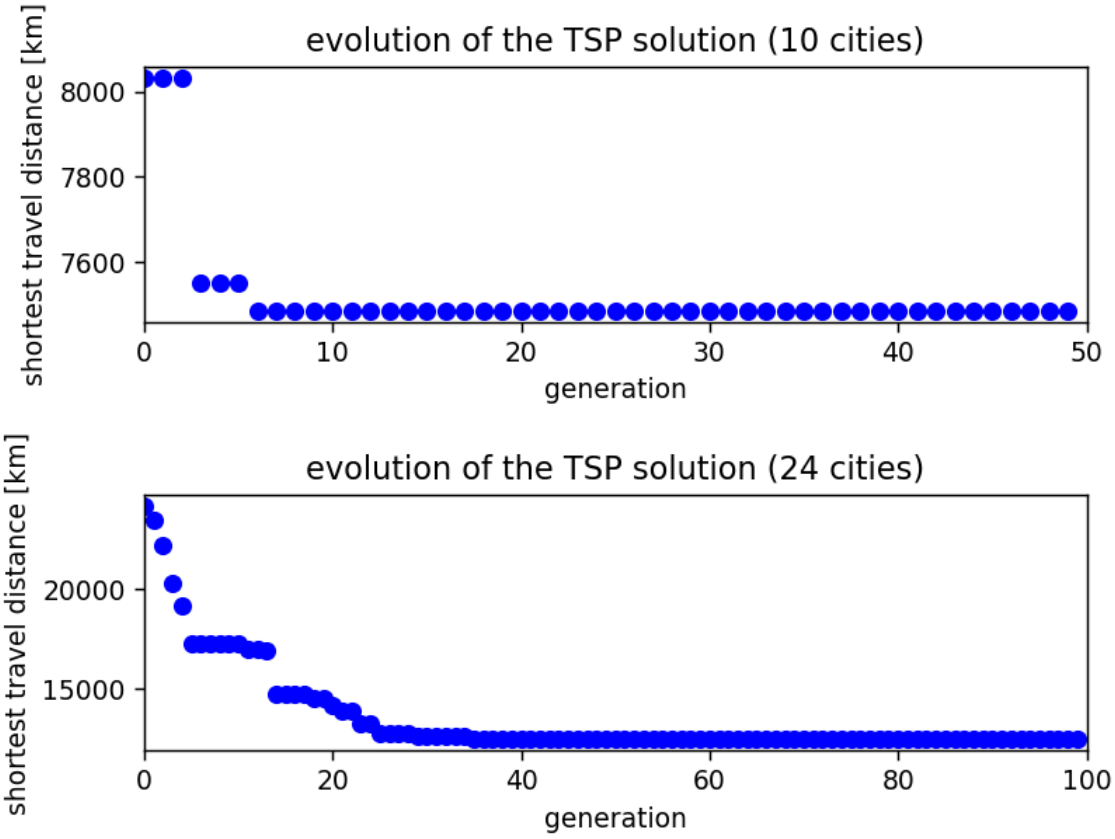
Q3.4) Among the first 10 cities, did your GA find the shortest tour (as found by the exhaustive search)? Did it come close?

In [14]:
```
fig, (ax1, ax2) = plt.subplots(2)
N = 10
tic = time.perf_counter()
tour_length = evolutution_TSP(distances[:N, :N], 100, 100,
                              50, rank_based_selection,
                              order_crossover_pair, elitism)
toc = time.perf_counter()
ax1.plot(np.arange(len(tour_length)), tour_length, "bo")
ax1.set_title("evolution of the TSP solution ({:.0f} cities)".format(N
))
ax1.set_xlim(0, 50)
print("run time: {:.3f} seconds".format(toc - tic))
print("shortest tour: {:.2f} km".format(tour_length[-1]))
print("shortest tour exhaustive search: {:.2f} km".format(7486.31))

N = 24
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
tic = time.perf_counter()
tour_length = evolutution_TSP(distances[:N, :N], 500, 500,
                              100, rank_based_selection,
                              order_crossover_pair, elitism)
toc = time.perf_counter()
ax2.plot(np.arange(len(tour_length)), tour_length, "bo")
ax2.set_title("evolution of the TSP solution ({:.0f} cities)".format(N
))
ax2.set_xlim(0, 100)
print("run time: {:.3f} seconds".format(toc - tic))
print("shortest tour: {:.2f} km".format(tour_length[-1]))
print("shortest tour exhaustive search: {:.2f} km".format(12287.07))

for ax in (ax1, ax2):
    ax.set_xlabel("generation")
    ax.set_ylabel("shortest travel distance [km]")
fig.tight_layout(pad=2.0)
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD




```
run time: 0.487 seconds
shortest tour: 7486.31 km
shortest tour exhaustive search: 7486.31 km
run time: 8.852 seconds
shortest tour: 12504.08 km
shortest tour exhaustive search: 12287.07 km
```

Yes! The shortest tour between 10 cities was found in generation 9.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Q3.5) For both 10 and 24 cities: How did the running time of your GA compare to that of the exhaustive search?

The runtime depends linearly on the number of generations and the population size. For the chosen values above, the runtime for 10 cities decreases from 23 seconds to less than 0.5 seconds. The runtime for 24 cities decreases from practically infinity to 9 seconds.

Q3.6) How many tours were inspected by your GA as compared to by the exhaustive search?

In [15]:
```
case_10 = 100 * 50
case_24 = 500 * 100
print("10 cities: {:.0f} out of {:.0f} tours inspected (= {:.2f} %)".fo
rmat(case_10, factorial(10), case_10/factorial(10)*100))
print("24 cities: {:.0f} out of {:.0f} tours inspected (= {:.8f} %)".fo
rmat(case_24, factorial(24), case_24/factorial(24)*100))
```

```
10 cities: 5000 out of 3628800 tours inspected (= 0.14 %)
24 cities: 50000 out of 620448401733239409999872 tours inspected (= 0.0
0000000 %)
```

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

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

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)

Q4) 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?

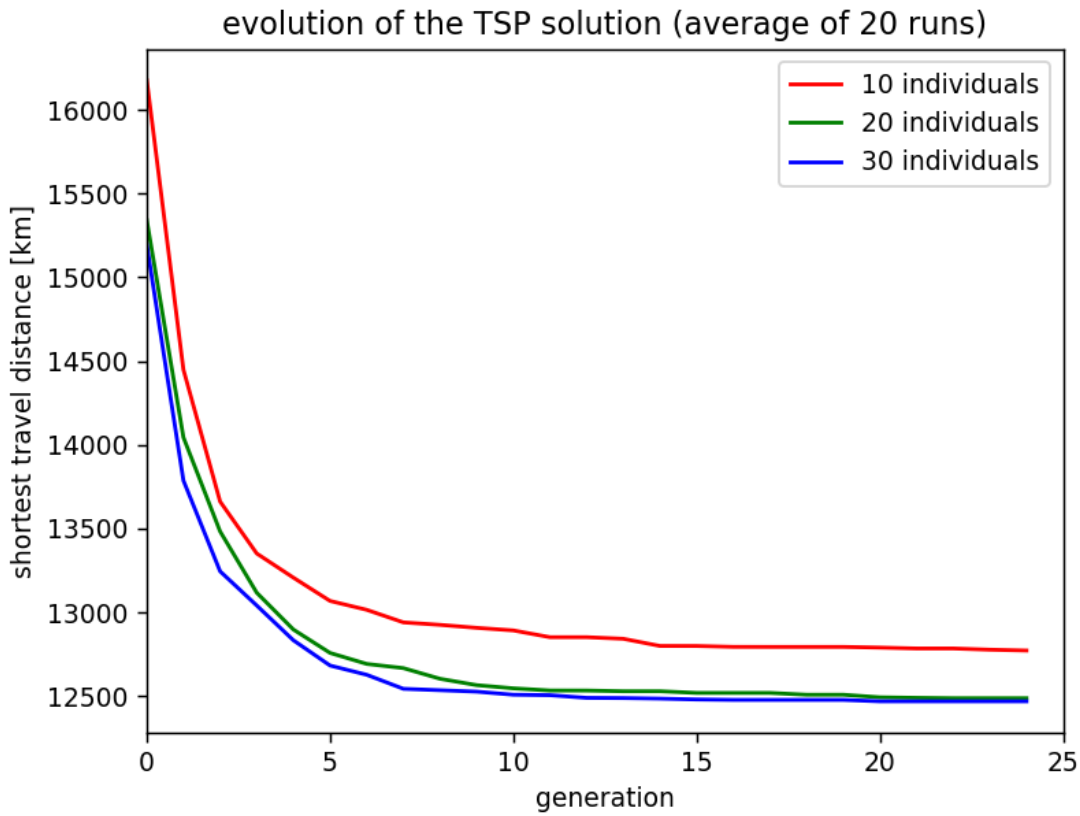
In [16]:
```
def test_hybrid(name):
    """Implement a hybrid algorithm to solve the TSP: Couple your GA an
d hill
    climber by running the hill climber a number of iterations on each
    individual in the population as part of the evaluation.
    Test both Lamarckian and Baldwinian learning models and report the
 results
    of both variants in the same way as with the pure GA (min, max, mea
n and
    standard deviation of the end result and an averaged generational p
lot).
    How do the results compare to that of the pure GA, considering the
 number
    of evaluations done?"""
    n_generations = 25
    n_runs = 20
    tour_length = np.empty(20, )
    fig, ax = plt.subplots()
    b = np.empty((3, n_runs, n_generations))
    x = np.arange(n_generations)
    colors = ["r", "g", "b"]
    for i, n_survivors in enumerate([10, 20, 30]):
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
        c = colors[i]
        n_parents = n_survivors
        print("number of individuals:", n_survivors)
        print("run: ", end="")
        for j in range(n_runs):
            print(j, end=", ")
            b[i, j] = evolutution_TSP(distances, n_survivors, n_parents
,
                                      n_generations, rank_based_selecti
on,
                                      order_crossover_pair, elitism,
                                      hybrid=name)  # Lamarck, Baldwini
an
            tour_length[j] = b[i, j, -1]
        print("""
        shortest tour: {:.2f} km
        longest tour: {:.2f} km
        average tour: {:.2f} km
        standard deviation {:.2f} km """.format(np.min(tour_length), np
.max(tour_length),
                                                np.mean(tour_length), n
p.std(tour_length)))
        y = np.mean(b[i], axis=0)
        plt.plot(x, y, c+"-", label="{:.0f} individuals".format(n_survi
vors))
    plt.legend()
    plt.xlim(0, n_generations)
    plt.xlabel("generation")
    plt.ylabel("shortest travel distance [km]")
    plt.title("evolution of the TSP solution (average of 20 runs)")
    plt.show()
```

In [17]:
```
test_hybrid("Lamarck")
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

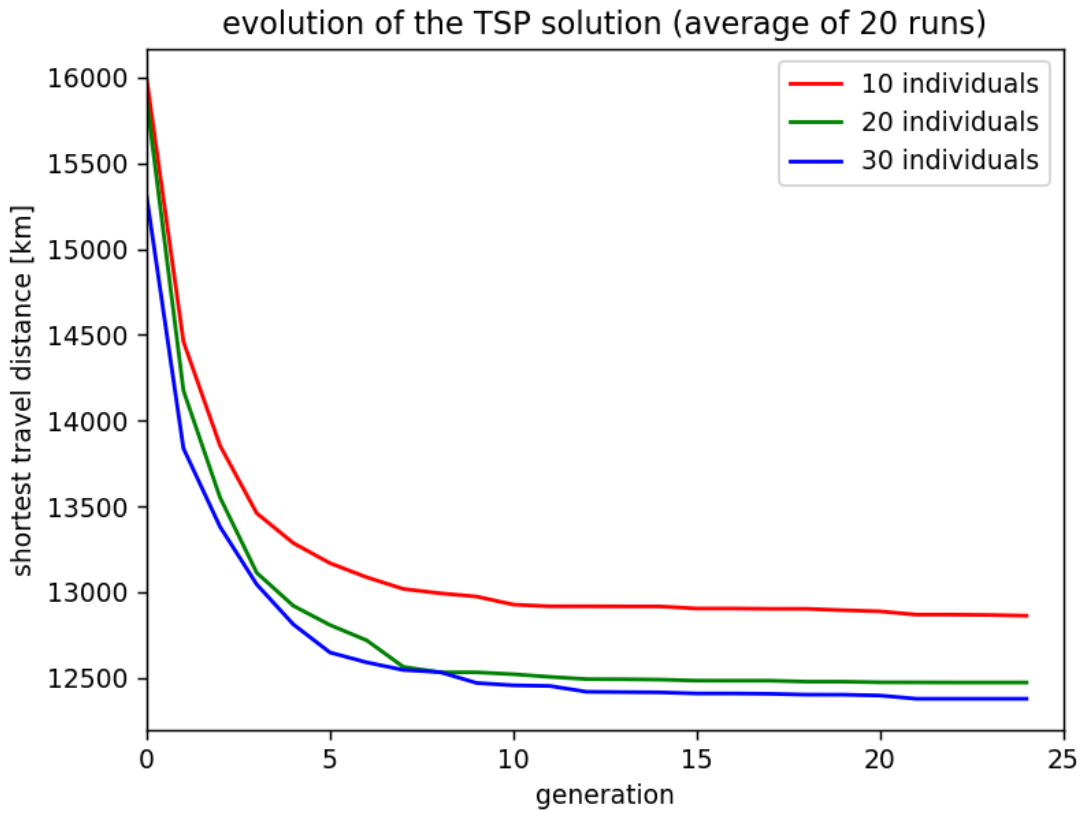



```
number of individuals: 10
run: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 1
8, 19,
        shortest tour: 12287.07 km
        longest tour: 13614.22 km
        average tour: 12772.12 km
        standard deviation 341.56 km
number of individuals: 20
run: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 1
8, 19,
        shortest tour: 12325.93 km
        longest tour: 12867.59 km
        average tour: 12487.54 km
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
        standard deviation 149.85 km
number of individuals: 30
run: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 1
8, 19,
        shortest tour: 12287.07 km
        longest tour: 12837.39 km
        average tour: 12468.79 km
        standard deviation 160.47 km
```

In [18]:
```
test_hybrid("Baldwinian")
```




```
number of individuals: 10
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
number of individuals: 10
run: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 1
8, 19,
        shortest tour: 12287.07 km
        longest tour: 13772.75 km
        average tour: 12862.60 km
        standard deviation 371.02 km
number of individuals: 20
run: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 1
8, 19,
        shortest tour: 12287.07 km
        longest tour: 13180.80 km
        average tour: 12473.09 km
        standard deviation 225.46 km
number of individuals: 30
run: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 1
8, 19,
        shortest tour: 12287.07 km
        longest tour: 12612.03 km
        average tour: 12378.20 km
        standard deviation 111.77 km
```

For each child, every possible swap of two cities is evaluated once. This means (24x24-24)/2 = 276 additional evaluations! In order to get results within a few minutes, the poulation size as wel as the number of generations had to be decreased accordingly. Previously there were 1000 generations and 200 individuals (200 tousand evaluations), now there are 25 generations and 30 individuals that get 276 times tested (207 thousand evaluations). As the standard deviation did not increase or decrease significantly compared to the pure GA, one must conclude, that the problem does not benefit from the hybrid solution.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD