# INF3490 Mandatory Assignment 1

Paul Wieland

Deadline: September 21, 2018

## Contents

# 1 General

This assignment consists of four other files. The european_cities.py, exhaustive_search.py, hill_climbing.py, genetic_algorithm.py and common.py. So for each algorithm is one file given that can be executed. How to use them will be explained later.
The common.py file is not meant to be executed.There are just some functions implemented that are used in several other files. For example the function *calculate_fitness(cities,data)* (line 14,common.py) calculates the total length of a specific route. How to use this functions is explained in the comments. So in fact this file should just avoid code redundancy.

**All these functions have been successfully tested on a Ubuntu Linux 18.04 machine executed with python3.**

Used libraries (to be sure that the files can be executed):

- timeit
- numpy
- random
- itertools
- sys
- matplotlib

# 2 Exhaustive Search

The following exhaustive search algorithm works very simple. It tries all possible combinations of how a set of cities can be visited. The actually implementation steps can be seen in the next subsection.

## 2.1 Code Overview

```
 9  def exhaustive_search(number_cities):
10      #import the data
11      data = import_data()
12      #get a set with cities with size of number_cities
13      cities = get_set_of_cities(number_cities, data)
14      #create a set with all possible permutations
15      permutations = list(iter.permutations(cities,number_cities))
16      #calculate the result
17      return iterate(permutations,data)
```

Core function of exhaustive search

The function that can be seen in the graphic above is the core function because it handles the logic to determine the best solution. There are just three steps to explain in detail.
In line 11 is the data imported from the csv file. In line 13 we can see that a

subset of cities is created that has the size of the parameter number_cities. In line 15 is the function permutations form the itertools used to determine all possible permutations. So the size of the variable permutations is exactly *number_cities!* (!: factorial). And last but not least is the function *iterate()* called that iterates over all permutations to determine their fitness using *calculate_fitness(cities, data)*. The best solution will be returned.

## 2.2 Use the program

### 2.2.1 Run the program

To run the program just execute the file:

*python3 exhaustive_search.py*

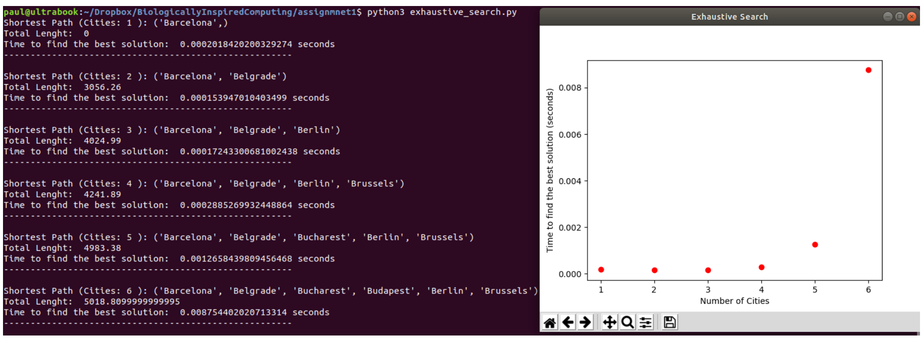
### 2.2.2 Change parameter

```
82 def main():
83     #change the number of cities
84     number_of_cities = 6
85     #used to determine several solutions and draw them
86     #output: console(shortest paths), pyplot drawing
87     plot_several_solutions(number_of_cities)
88
89     #prints one certain solution on the console (uncomment to use it)
90     #--------------------------------
91     #single_solution(number_of_cities)
92     #--------------------------------
```

Main function of exhaustive search

To change the number of cities that should be visited you can modify the parameter *number_of_cities* to a number higher than zero.
Per default is the function *plot_several_solutions()* called. This function determines the best solution for different number of cities. All solutions are going to be printed to the console. This function creates also a graphical representation that shows the number of cities in terms to the run-time.

It should look like this:




Output calling the function *plot_several_solutions(6)*

But it is also possible to determine the solution for just one certain number of cities. For this you have to switch the comments in line 87 and line 91. So that the function *single_solution(number_of_cities)* is called. This function prints the solution to the console as you can see in 2.3.1.

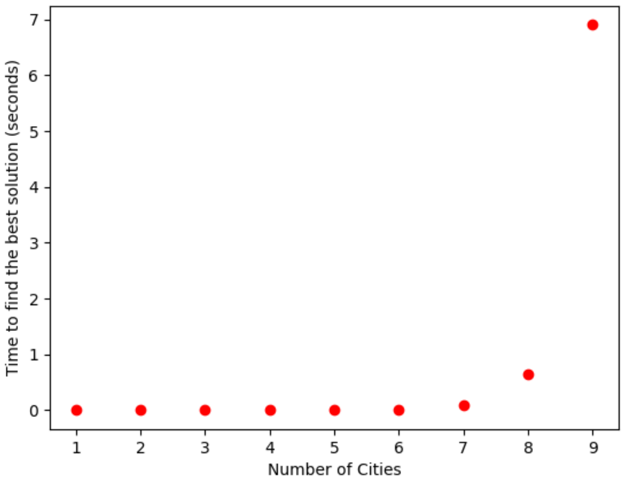
## 2.3 Questions

### 2.3.1 Subset of 6 cities

```
Shortest Path (Cities: 6 ): ('Barcelona', 'Belgrade', 'Bucharest', 'Budapest', 'Berlin',
 'Brussels')
Total Lenght:  5018.8099999999995
Time to find the best solution:  0.007981712988112122 seconds
```

Shortest path (subset of 6 cities)

### 2.3.2 Incrementally add more cities

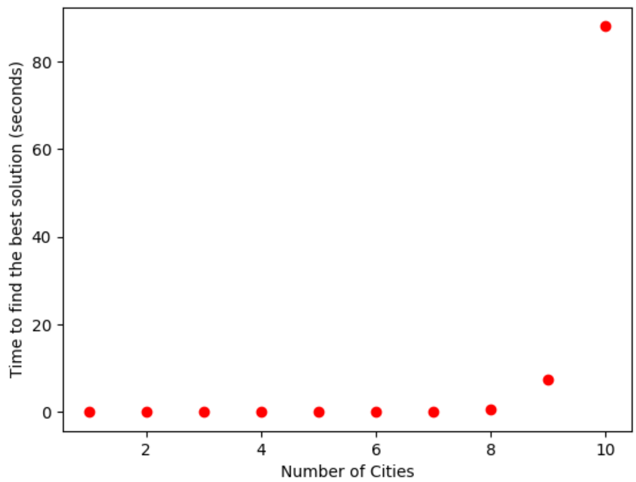



The affect of adding more cities can bee seen very well in the graphic above. The run-time increases exponentially with the number of cities.

### 2.3.3 Subset of 10 cities

```
Shortest Path (Cities: 10 ): ('Copenhagen', 'Hamburg', 'Brussels', 'Dublin', 'Barcelona'
, 'Belgrade', 'Istanbul', 'Bucharest', 'Budapest', 'Berlin')
Total Lenght:  7486.309999999999
Time to find the best solution:  84.42702381600975 seconds
```

Shortest Path (subset of 10 cities)




Graphical Representation

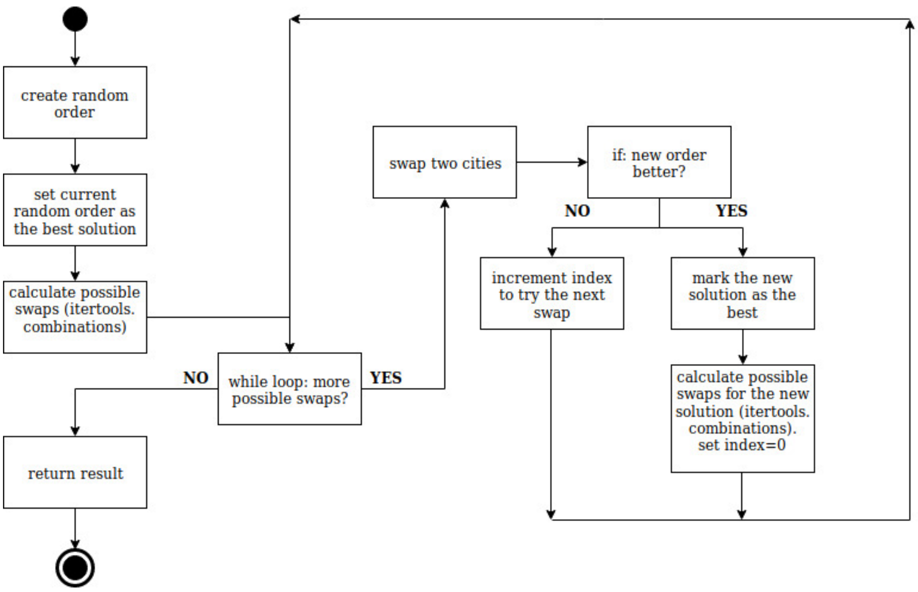
#### 2.3.4 Subset of 24 cities (expectation)

While a subset of 10 cities has 10! (3,628,800) possible solutions, a set with 24 cities has 620,448,401,733,239,439,360,000 possible solutions. So it is not possible to solve this in human live. It would take more than $4 \times 10^{11}$ years.

# 3 Hill Climbing

The following hill-climbing implementation firstly creates a random order. To improve this initial solution, the algorithm swaps two cities an check if there is an improvement. A more detailed description can be seen in 3.1 and in the file *hill_climbing.py*.

## 3.1 Code Overview




Graphical representation of the function *hill_climbing(number_of_cities)*

The *hill_climbing(number_of_cities)* function is the heart of the hill-climber program. It just follows the steps you can see above in the flowchart.

## 3.2 Use the program

### 3.2.1 Run the program

To run the program just execute the file:

*python3 hill_climbing.py*

### 3.2.2 Change parameter

```
def main():
    #number of cities
    number_of_cities = 24
    #number of runs that should be done
    number_of_tests = 20
    print_hill_climber(number_of_cities, number_of_tests)
```

Main function of *hill_climbing.py*

number_of_cities: Number of cities that should be visited
number_of_tests : Number of runs that should be made

Per default is the function *print_hill_climber(number_of_cities, number_of_tests)* called. The result of calling this function should look like 3.3.2 or 3.3.3.

## 3.3 Questions

### 3.3.1 Compare hill climbing to exhaustive search (10 cities)

```
Shortest Path (Cities: 10 ): ('Copenhagen', 'Hamburg', 'Brussels', 'Dublin', 'Barcelona'
, 'Belgrade', 'Istanbul', 'Bucharest', 'Budapest', 'Berlin')
Total Lenght:  7486.309999999999
Time to find the best solution:  84.42702381600975 seconds
```

Exhaustive Search, Shortest Path (subset of 10 cities)

```
Average Time has been determined with 20 tests.
Shortest Path (Best case) (Cities 10 ): ['Berlin', 'Budapest', 'Bucharest', 'Istanbul',
'Belgrade', 'Barcelona', 'Dublin', 'Brussels', 'Hamburg', 'Copenhagen']
Average Time:            0.004365245599183254  seconds
Best case:               7486.309999999999
Worst case:              8529.929999999998
Average:                 7708.525
Standard deviation:      297.16883387226176
```

Hill Climbing Result, 10 Cities, 20 runs

As we can see, the hill climbing algorithm is much faster. For 10 cities:

| | |
|---|---|
| Exhaustive Search: | 82.4270 seconds |
| Hill-Climbing (Average): | 00.0043 seconds |

### 3.3.2 Result 10 Cities, 20 runs


```
Average Time has been determined with 20 tests.
Shortest Path (Best case) (Cities 10 ): ['Berlin', 'Budapest', 'Bucharest', 'Istanbul',
'Belgrade', 'Barcelona', 'Dublin', 'Brussels', 'Hamburg', 'Copenhagen']
Average Time:            0.004365245599183254  seconds
Best case:               7486.309999999999
Worst case:              8529.929999999998
Average:                 7708.525
Standard deviation:      297.16883387226176
```

Hill Climbing Result, 10 Cities, 20 runs


### 3.3.3 Result 24 Cities, 20 runs

```
Average Time has been determined with 20 tests.
Shortest Path (Best case) (Cities 24 ): ['Madrid', 'Barcelona', 'Milan', 'Rome', 'Belgra
de', 'Sofia', 'Istanbul', 'Bucharest', 'Budapest', 'Vienna', 'Warsaw', 'Kiev', 'Moscow',
 'Saint Petersburg', 'Stockholm', 'Copenhagen', 'Hamburg', 'Berlin', 'Prague', 'Munich',
 'Paris', 'Brussels', 'London', 'Dublin']
Average Time:            0.14614353170036337  seconds
Best case:               12709.199999999997
Worst case:              20707.57
Average:                 15708.240499999996
Standard deviation:      2275.581242976121
```

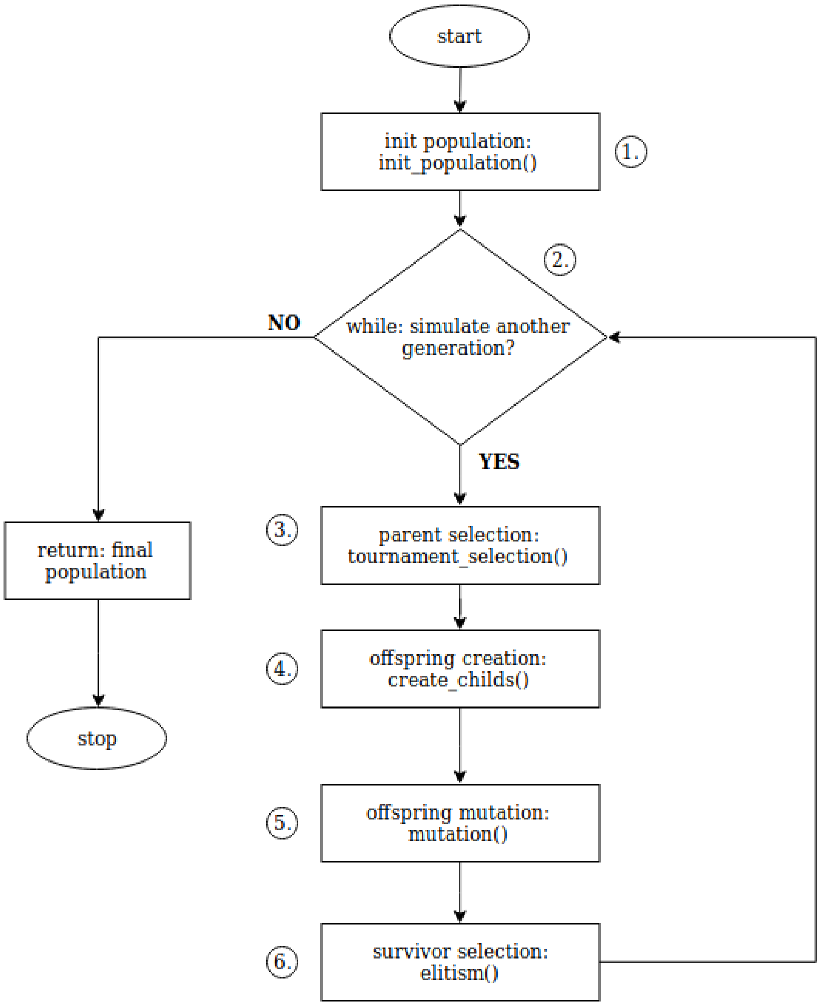
Hill Climbing Result, 24 Cities, 20 runs

# 4 Genetic Algorithm

## 4.1 General

| | |
|---|---|
| Representation: | List of destinations (Adjacency Representation) |
| Recombination: | Partially Mapped Crossover (PMX) |
| Mutation: | Inversion Mutation (Because of the adjacency problem) |
| Parent Selection: | Tournament Selection (Fitness-Proportionate-Selection) |
| Survivor Selection: | Elitism, preserve the N best individuals. Other randomly(Exploration) |

## 4.2 Code overview




Graphical representation of the main procedure of *genetic_algorithm.py*

The graphic above show the fundamental procedure of the *genetic_algorithm* program. There are several steps in this program, i would like to explain in detail.

1. Initialization of the Population:
   Firstly, a certain size of individuals will be created randomly to initialize the population.

2. Loop:
   This loop provides a simulation of several generations.

3. Parent Selection:
   This function selects a number of individuals that are able to create new children. The selection is made by a Tournament-Selection. There are two parameter that steers the selection. The first one is the *number_of_tournament_winners* that says how many parents should win the

tournament. The second parameter is the *tournament_size* that determines the number of individuals that are compared to each other (This parameter steers the *selection pressure*). The best individual will be selected. The tournament is created randomly.

4. Offspring Creation:
   The tournament winners can now create new children. Each winner-parent can create children with any other winner-parent. To determine the combinations the function *combinations()* from *itertools* is used. Each combination will be executed with a certain *crossover_rate*. The crossover will be done by the pmx-function that you can find in the file *PMX.py* (Because of the adjacency-based problem).

5. Offspring Mutation:
   Each offspring will be modified with a certain *permutation_rate*. The inversion-mutation is used because it breaks only two links. That is an advantage for our kind of problem.

6. Survivor Selection:
   The survivor selection follows the *elitism* principle. The $N$ best individuals will be selected for the next generation. The other *populations_size* - $N$ individuals will be selected randomly, because for the reason of Exploration. Without this random selection, the algorithm converges very fast.

## 4.3 Use the program

### 4.3.1 Run the program

To run the program just execute the file:

*python3 genetic_algorithm.py*

### 4.3.2 Change parameter

```
#parameter:
#number_of_cities                (eg 6,10,24)
number_of_cities = 10
#elite                           ('elite' best individuals will be preserved for the next round)
elite = 3
#popultation_size                (number of individuals )
popultation_size = 10
#crossover_rate                  (probability to crossover two parents, create two new children)
crossover_rate = 0.5
#mutation_rate                   (probability of an inversion_mutation)
mutation_rate  =0.5
#generations                     (number of generations that should be simulated)
generations = 10
#tournament_size                 (size of the tournaments, higher number means higher selection pressure)
tournament_size = 2
#number_of_tournement_winners   (size of the parents that do crossover, after parent selection)
number_of_tournement_winners = 10
```

Parameter that can be changed in the file *genetic_algorithm.py*.

## 4.4 Results of changing parameter

### 4.4.1 Crossover Rate, Permutation Rate

```
----------Genetic Algorithm----------
Best individual of each Generation is basic of this statistic.
Average Time has been determined with 20 tests.
Shortest Path (Best case) (Cities 10 ): ['Bucharest', 'Budapest', 'Copenhagen', 'Berlin', 'Hamburg',
'Barcelona', 'Dublin', 'Brussels', 'Belgrade', 'Istanbul']
Runtime:                 0.3597873899998376 seconds
Best case:               7780.629999999999
Average case:            8769.9205
Worst case:              10167.03
Standard deviation:      603.3005003020887
```

Permutation Rate and Crossover Rate: 10% (0.1)

```
----------Genetic Algorithm----------
Best individual of each Generation is basic of this statistic.
Average Time has been determined with 20 tests.
Shortest Path (Best case) (Cities 10 ): ['Brussels', 'Dublin', 'Barcelona', 'Belgrade', 'Istanbul', '
Bucharest', 'Budapest', 'Berlin', 'Copenhagen', 'Hamburg']
Runtime:                 1.680671746000371 seconds
Best case:               7486.3099999999995
Average case:            7682.627
Worst case:              8323.44
Standard deviation:      286.08107550308216
```

Permutation Rate and Crossover Rate: 90% (0.9)

More Mutations and Crossover means a better result but also more time to do this operations.

### 4.4.2 Number of Elite Survivor

```
----------Genetic Algorithm----------
Best individual of each Generation is basic of this statistic.
Average Time has been determined with 20 tests.
Shortest Path (Best case) (Cities 10 ): ['Barcelona', 'Brussels', 'Berlin', 'Hamburg', 'Budapest', 'B
ucharest', 'Istanbul', 'Belgrade', 'Copenhagen', 'Dublin']
Runtime:                 0.39217426399955 seconds
Best case:               7549.16
Average case:            8260.485
Worst case:              9063.25
Standard deviation:      392.2588675925631
```

10% of the best children will be kept.

```
----------Genetic Algorithm----------
Best individual of each Generation is basic of this statistic.
Average Time has been determined with 20 tests.
Shortest Path (Best case) (Cities 10 ): ['Belgrade', 'Bucharest', 'Istanbul', 'Barcelona', 'Brussels'
, 'Dublin', 'Hamburg', 'Copenhagen', 'Berlin', 'Budapest']
Runtime:                 2.8751846100003604 seconds
Best case:               7486.3099999999995
Average case:            7629.1685
Worst case:              7915.34
Standard deviation:      123.01475733728073
```

90% of the best children will be kept.

The more better solutions are kept, the more better is the result. But that means also a much smaller deviation. That can easily result in local minimum because the solutions are very similar.

### 4.4.3 Number of Generations

```
----------Genetic Algorithm----------
Best individual of each Generation is basic of this statistic.
Average Time has been determined with 20 tests.
Shortest Path (Best case) (Cities 10 ): ['Hamburg', 'Copenhagen', 'Berlin', 'Bucharest', 'Istanbul',
'Belgrade', 'Budapest', 'Barcelona', 'Dublin', 'Brussels']
Runtime:                 1.020624945000236 seconds
Best case:               7503.1
Average case:            7848.043000000001
Worst case:              8606.19
Standard deviation:      236.01268139869103
```

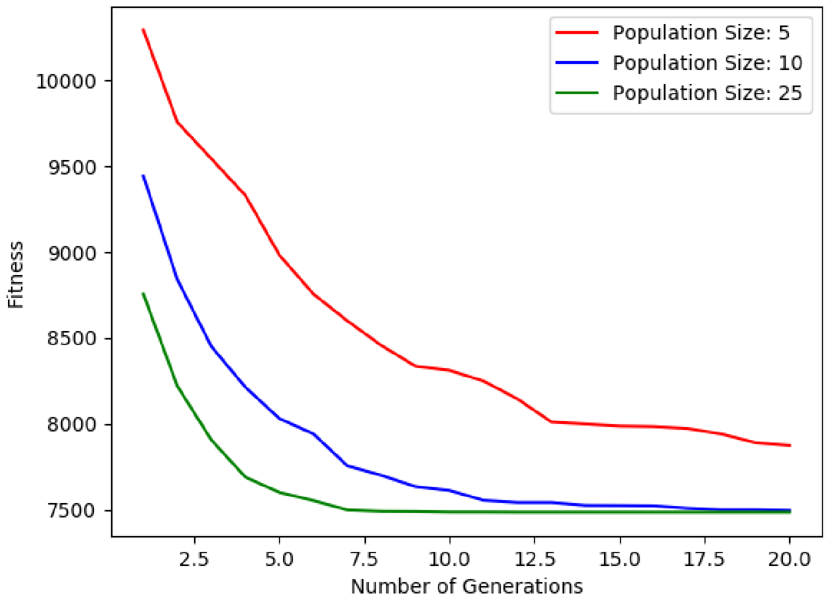
10 Generations

```
----------Genetic Algorithm----------
Best individual of each Generation is basic of this statistic.
Average Time has been determined with 20 tests.
Shortest Path (Best case) (Cities 10 ): ['Barcelona', 'Belgrade', 'Istanbul', 'Bucharest', 'Budapest'
, 'Berlin', 'Copenhagen', 'Hamburg', 'Brussels', 'Dublin']
Runtime:                 9.936407384000631 seconds
Best case:               7486.309999999999
Average case:            7486.3099999999995
Worst case:              7486.31
Standard deviation:      5.380643217995035e-13
```

100 Generations

The more generations there are, the better is the result of course. I think is the best way to improve the result. But it also takes much more time for the result.

#### 4.4.4 Size of Population, 3 different sizes (Average across runs)




20 runs, 3 different sizes of population, average of best individual

The bigger the population size is, the faster the algorithm converges. A reason for this is that a bigger population has from the beginning more better solution. That makes sense because the population is created randomly.
As elitism is used, it is more likely that better solutions create new offspring. So that is why the biggest population (green line) has from the beginning very good individuals and converges very fast.

### 4.4.5 Tournament size

```
----------Genetic Algorithm----------
Best individual of each Generation is basic of this statistic.
Average Time has been determined with 20 tests.
Shortest Path (Best case) (Cities 10 ): ['Bucharest', 'Budapest', 'Copenhagen', 'Berlin', 'Hamburg',
'Brussels', 'Dublin', 'Barcelona', 'Belgrade', 'Istanbul']
Runtime:                 1.0154521059994295 seconds
Best case:               7486.3099999999995
Average case:            7834.862000000001
Worst case:              8486.650000000001
Standard deviation:      316.39177262691294
```

Tournament size of 2.

```
----------Genetic Algorithm----------
Best individual of each Generation is basic of this statistic.
Average Time has been determined with 20 tests.
Shortest Path (Best case) (Cities 10 ): ['Copenhagen', 'Berlin', 'Budapest', 'Bucharest', 'Istanbul',
 'Belgrade', 'Barcelona', 'Dublin', 'Brussels', 'Hamburg']
Runtime:                 1.0261872600003699 seconds
Best case:               7486.3099999999995
Average case:            7899.349499999999
Worst case:              8456.3
Standard deviation:      289.88261678608814
```

Tournament size of 8.

A higher tournament size means a better population quality (lower deviation). Because the selection pressure is higher.

### 4.4.6 Number of tournament winner

```
----------Genetic Algorithm----------
Best individual of each Generation is basic of this statistic.
Average Time has been determined with 20 tests.
Shortest Path (Best case) (Cities 10 ): ['Bucharest', 'Copenhagen', 'Berlin', 'Hamburg', 'Brussels',
'Belgrade', 'Dublin', 'Barcelona', 'Budapest', 'Istanbul']
Runtime:                 0.17140544100038824 seconds
Best case:               8592.33
Average case:            10284.3075
Worst case:              11454.43
Standard deviation:      744.5485503100183
```

10% of the parents win the tournament.

```
----------Genetic Algorithm----------
Best individual of each Generation is basic of this statistic.
Average Time has been determined with 20 tests.
Shortest Path (Best case) (Cities 10 ): ['Berlin', 'Hamburg', 'Copenhagen', 'Brussels', 'Dublin', 'Ba
rcelona', 'Istanbul', 'Bucharest', 'Belgrade', 'Budapest']
Runtime:                 0.947320311999647 seconds
Best case:               7486.31
Average case:            7977.960999999998
Worst case:              8943.880000000001
Standard deviation:      370.64676008161763
```

90% of the parents win the tournament.

The more parents win the tournament, the more offspring will be created. That means also more diversity and a higher probability for a good solution.

## 4.5 Questions

### 4.5.1 Result 6 Cities, 20 runs

```
----------Genetic Algorithm----------
Best individual of each Generation is basic of this statistic.
Average Time has been determined with 20 tests.
Shortest Path (Best case) (Cities 6 ): ['Bucharest', 'Belgrade', 'Barcelona', 'Brussels', 'Berlin', '
Budapest']
Runtime:                 1.082016084001225 seconds
Best case:               5018.8099999999995
Average case:            5018.8099999999995
Worst case:              5018.8099999999995
Standard deviation:      0.0
```

20 Generations, 6 Cities

### 4.5.2 Result 10 Cities, 20 runs

```
----------Genetic Algorithm----------
Best individual of each Generation is basic of this statistic.
Average Time has been determined with 20 tests.
Shortest Path (Best case) (Cities 10 ): ['Bucharest', 'Istanbul', 'Belgrade', 'Budapest', 'Berlin', '
Copenhagen', 'Hamburg', 'Brussels', 'Dublin', 'Barcelona']
Runtime:                2.4005894209985854 seconds
Best case:              7486.309999999999
Average case:           7602.812
Worst case:             8277.8
Standard deviation:     188.24937844784515
```

20 generations, 10 Cities

| Algorithm | Runs | Best solution | Runtime | Compared tours |
|---|---|---|---|---|
| Exhaustive Search | 20 | 7486 | 84.4 s | 3,628,800 |
| Genetic Algorithm | 20 | 7486 | 2.4 s | 100* |

100*: 10 Generations with population size of 10
We can see, that the GA finds also the best solution in a much shorter time.

### 4.5.3 Result 24 Cities, 20 runs

```
----------Genetic Algorithm----------
Best individual of each Generation is basic of this statistic.
Average Time has been determined with 20 tests.
Shortest Path (Best case) (Cities 24 ): ['Prague', 'Vienna', 'Belgrade', 'Sofia', 'Istanbul', 'Buchar
est', 'Budapest', 'Milan', 'Munich', 'Rome', 'Barcelona', 'Madrid', 'Paris', 'Dublin', 'London', 'Cop
enhagen', 'Stockholm', 'Saint Petersburg', 'Moscow', 'Kiev', 'Warsaw', 'Berlin', 'Hamburg', 'Brussels
']
Runtime:                28.258468356994854 seconds
Best case:              12805.810000000003
Average case:           14244.800500000001
Worst case:             16238.49
Standard deviation:     830.9093983490318
```

100 generations, 24 Cities

| Algorithm | Runs | Best solution | Runtime | Compared tours |
|---|---|---|---|---|
| Exhaustive Search | 20 | 7486 | -** | $6.2 \times 10^{23}$ |
| Genetic Algorithm | 20 | 12,805 | 28.3 s | 1000* |

1000*: 100 Generations with population size of 10
-**: Not possible to compute