Evolutionary Algorithms: Final report

Miguel Quirijnen (r0710304)

June 23, 2022

1 Metadata

- **Group members during group phase:** Julie Braeckevelt, Miguel Quirijnen, Simon Catteau and Winand Appels
- **Time spent on group phase:** 11 hours
- **Time spent on final code:** 38 hours
- **Time spent on final report:** 12 hours

2 Peer review reports

2.1 The weak points

1. The **initialization** of the population was **completely random**. A more advanced, e.g. local search, mechanism was suggested. One of the main arguments to start with a better population, is the time limit of five minutes.
2. The **static parameters** do not take the evolution of the population, nor the population size, nor the problem size into account. A replacement of these with (self-)adaptive parameters would be a better fit for the algorithm.
3. To address the issue around quick convergence, one should **improve diversity**. This would avoid getting stuck in a local minimum. Introducing bigger jumps in the search space are suggested.
4. The **population and offspring size are equal**. An offspring size bigger than the population size should be considered.
5. The usage of a **stronger elimination operator** would be worth testing. They suggest using the k-tournament from the selection operator to test with.
6. **Lowering** the value of the **parameter k** in k-tournament would lead to good results when more exploration is necessary.

2.2 The solutions

1. The objective values in a small portion of the **initialized population are enhanced** by applying a heuristic on them. This addresses the time limit concern and gives the algorithm a head start. I experimented with different heuristics and all of them resulted in better individuals then a completely random initial population.
2. Most of the static parameters are replaced with **adaptive parameters**. These parameters depend on the current state or configuration of the algorithm.
3. The quick convergence of the algorithm is addressed by introducing **more diversity methods**. These make sure the algorithm does not get stuck in local minima or misses out on better solutions.
4. Due to the lack of diversity in the group phase, the **elimination operator was replaced**. This also adresses the early convergence problem.

1. The **population size** is kept **equal to the offspring size**. This was chosen after several experiments and a grid search.
2. I did not lower the **value of k in the k-tournament operator**. The argument given that a lower k leads to more exploration is definitely true, but since it was already low, it did not improve the solutions by lowering it even further.

2.3 The best suggestion

The best suggestion from the peer reviews was the one on **better initialization**. In the group phase the initialization used, was completely random. As already mentioned in the peer review, this works really well for smaller datasets (such as tour29 and tour100, which we used in the group phase). However, when increasing these datasets to sizes from 500 up to 1000 cities, this way of initializing does not suffice, especially considering the five minute time limit. Initializing some individuals based on local search gives the algorithm a head start, even if this is only for a small percentage of the initial population. Applying a **local search operator** to this **small portion of the randomly initialized population** improves their objective value significantly and lowers the total amount of iterations to reach a certain mean value.

3 Changes since the group phase

1. In the initialization phase, the algorithm now uses a local search operator, namely **nearest neighbors**. As already mentioned in the previous section, this significantly enriches the initial population and helps the solutions direct to a minimum. To make sure these optimized individuals do not take over the population, the nearest neighbor operator is only applied to a small portion.
2. Furthermore, a **2-opt local search** is added to the algorithm. After the mutation, every individual of the offspring is improved with local search optimization. This local search operator replaces an individual with the best solution in its second neighborhood.
3. Since these two first changes push the algorithm towards the exploitation side of the spectrum, improvement in diversity and prevention of premature convergence are necessary. Because of these reasons, the **population** is divided into **two islands**. These islands both use a different recombination mechanism and thus create a different offspring.
4. A **third mutation operator**, inversion mutation, was added to shift the algorithm even further to exploration. This allows the algorithm to improve the randomness of its offspring. Inversion mutation keeps most of the information of the parents and thus does not interfere the good performing individuals too much.
5. A second recombination mechanism, **Sequential Constructive Crossover (CSX)** is added to the algorithm. It is used in on a separate island as the first mechanism. This way, two populations can evolve nearly separately and leave room for a sufficient amount of diversity in the combined population.

4 Final design of the evolutionary algorithm

4.1 The three main features

1. Nearest Neighbour heuristic in initializing the population (Section 4.4)
2. Two-opt permutation as local search operator on the offspring (Section 4.9)
3. Dividing the population is different islands (Section 4.10)

4.2 The main loop

Figure 1 describes the flow and includes all the main components of the evolutionary algorithm. The components of this design are extensively discussed in the following sections.

4.3 Representation

The most natural and compact representation for the Traveling Salesman Problem is **path representation** [5]. The representation is optimal regarding convergence and in general gives good results. This means it will also be convenient for crossover and mutation operations, which will be discussed later. Other possible options are the binary, adjacency or ordinal representation. They are not considered because the representation is one of the fundamental design choices of the algorithm and it is important to make them **intuitive**.

In Python, the chosen path representation can be represented as an array of integers. These integers represent the city that is visited and the index of these integers represent the order. In python arrays start with index 0, this has to be kept in mind when using this representation (e.g. route[3] = 6 means that the fourth city visited is city number 6). To be more precise, a **Numpy array** was used to represent the path. In the group phase the route was represented as a list, but since the distance matrix was implemented as a numpy array and the 'numba' [7] compiler prefers this over a list, I changed it. This way there is more uniformity in the used types.

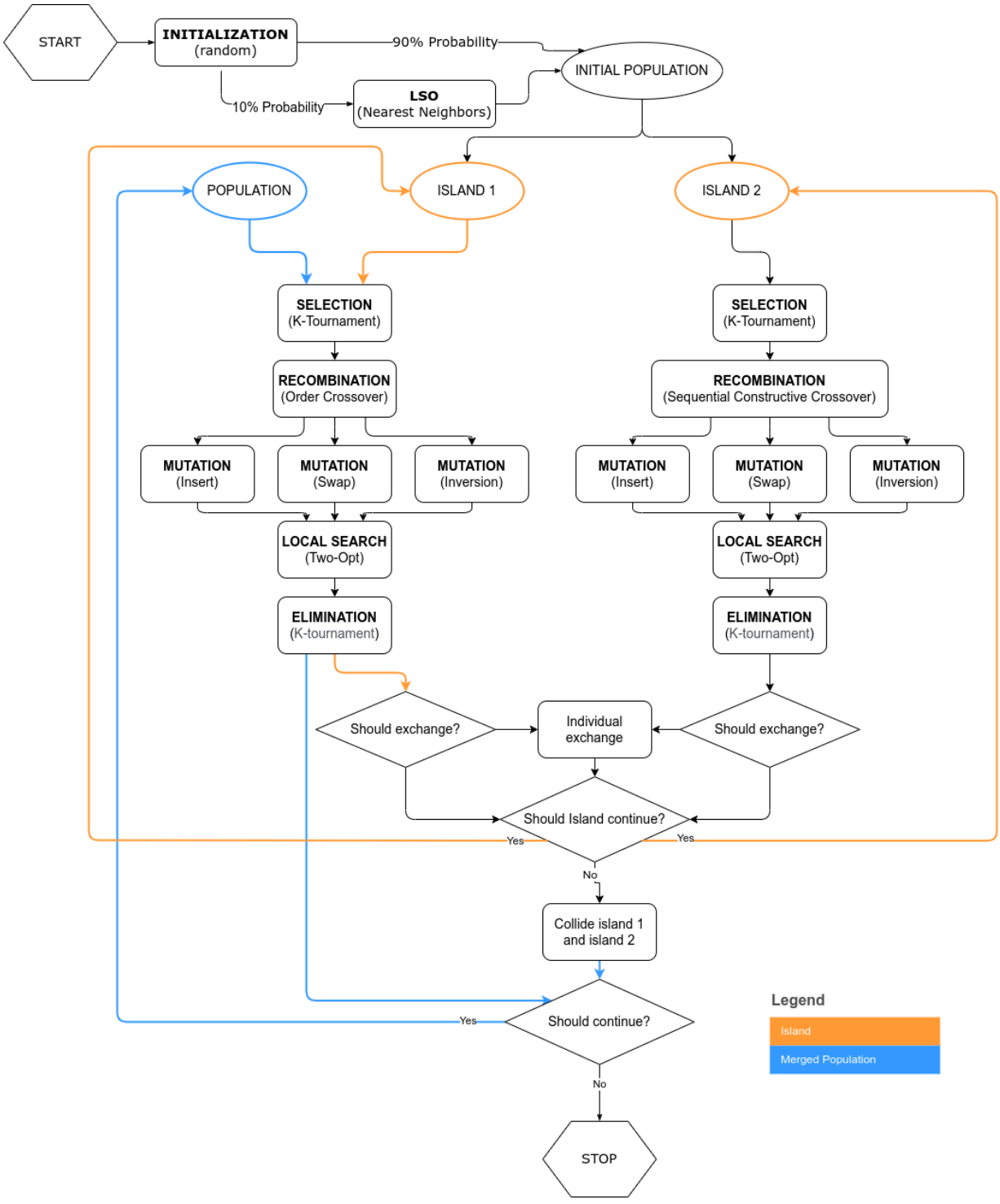

Figure 1: The main loop of the evolutionary algorithm

4.4 Initialization

The population is initialized in a loop. In this loop an individual is **randomly generated** and undergoes a **heuristic optimization** with a predefined probability. The number of individuals, λ, is determined experimentally and dependent on the problem size (the amount of cities, c). For larger problems, the population size is smaller. The main argument to do this is balancing out the computational cost per iteration. The usage of a heuristic optimization allows to achieve the same performance with a smaller population. This is beneficial since a larger population means a higher computational cost.

$$\lambda = \begin{cases} 180 & \text{if } c < 333; \\ 120 & \text{if } 333 <= c < 666; \\ 60 & \text{if } 666 <= c. \end{cases} \tag{1}$$

The local search heuristic that is applied, is the **nearest neighbor** algorithm. This algorithm takes the first node in the route of an Individual and puts the nearest unvisited node after it. Then it takes the added node and again finds the nearest unvisited node. It does this until there are no cities left and the route is complete.

By keeping the percentage of individuals going through the nearest neighbor algorithm relatively low (later discussed in parameter selection 4.12), this scheme does not immediately take over the population and most of the **diversity created by the randomness is kept**. Initially, the local search operator was applied to a much larger portion of the population. When testing this several times, it seemed that diversity suffered a lot under this condition or it was computationally to expensive.

4.5 Selection operators

The selection operator implemented is **k-tournament**. I did not make any changes to the operator itself since the group phase. The main goal of the selection operator is to return individuals with a high fitness, while preserving diversity. The only parameter that needs tuning in k-tournament is k. The parameter k can be tuned to **balance between exploration and exploitation**. Lowering k means lowering the selection pressure and putting emphasis on diversity (e.g. k=1 is equal to a pure random selection), increasing k results in increasing selection pressure. No advanced scheme was used to vary this parameter, it was chosen with a grid search (section 4.12). The amount of individuals selected in total is the offspring, μ. The offspring size is determined by a static ratio with the population size.

4.6 Mutation operators

To create the necessary diversity, the algorithm contains three mutation operators: **swap mutation, insert mutation and inversion mutation**. Each of these have positive and negative properties in the context of the Traveling Salesman Problem.

Swap mutation introduces minimal change to the cycles. The same goes for insert mutation. Inversion mutation is proven to perform well in adjacency problems like this project and introduces more randomness than swap mutation. Since there is randomness in the choice of the already random mutation operators, I believe that there is enough randomness to introduce a sufficient amount of diversity in the population.

The probability of mutation and the probabilities for each mutation operator, are computed by **adaptivity**. The probability of mutation, α, linearly decreases with every iteration. To control the general properties of the mutation, the **ratio's of the probabilities of the mutations are adapted taking into account their individual properties**. To be more precise: Insert and swap mutation are less aggressive mutation operators than inversion mutation. This is why I introduced adaptive parameters that make these two first operators less probable in the first iterations and make them dominant when the algorithm evolves. This results in a heavy diversion in the beginning phases, while only doing slight modifications when the population is more evolved and optimized.

$$\alpha = 0.9 - 0.8 * \frac{iteration}{max\_its} \tag{2}$$

$$p_{inversion} = 0.8 - 0.6 * \frac{iteration}{max\_its} \tag{3}$$

$$p_{swap} = 0.1 + 0.3 * \frac{iteration}{max\_its} \tag{4}$$

$$p_{insertion} = 1 - p_{inversion} - p_{swap} \tag{5}$$

4.7 Recombination operators

In the group phase, our team already implemented order crossover recombination and in the individual phase I added Sequential Constructive Crossover (SCX).

Order crossover is proposed by Davis [5] and is widely known for its proven performance in Traveling Salesman Problems. It transmits information about the relative order from the second parent. This relative order is important, because it determines the fitness of the individual.

While, order crossover does not take the distance matrix in account, **Sequential Constructive Crossover** does [3]. Using the **distance matrix** to count the cheapest edge can be advantageous in convergence. **SCX** starts with the first node of the first parent and starts searching for the next legitimate node in both parents (legitimate means not-visited or not out of bounds). If it is out of bounds, then it takes the next node in an ordered set of unvisited indices. It compares the cost of the step from the previous node to the next one in both parents and takes the cheapest one. This means it keeps a lot of the edges of the parents while improving the overall fitness, since it always takes the cheapest edge of the two. If the parents already had an overlap, this is maintained.

The recombination is not controlled with parameters, but instead it is **divided over two islands** (Section 4.10). Recombination with arity greater than two was not considered. It would make the algorithms more complex and computationally expensive ($O(N^2)$ instead of $O(N)$) than an arity of exactly two. The benefit of this would not weigh up against the extra computational cost.

4.8 Elimination operators

As elimination operators I re-used **k-tournament** from the selection operator. It takes the individual with the best fitness (the current best value) and adds $\lambda - 1$ individuals selected with k-tournament. This way, the amount of survivors is equal to the original population size, λ. In doing this, the algorithm **maintains a diverse population** while also keeping the best individual and **introducing selective pressure**. The value of the parameter k is the same as in the selection operator. I also tried to use $(\lambda + \mu)$ as elimination operator, which was used in the group phase of the project. This operator was discarded since it had a negative impact on the diversity in the population. The mean value really quickly converged to the best value when using $(\lambda + \mu)$.

4.9 Local search operators

I have implemented two local search operators, namely **two-opt** and **nearest neighbor**. The general idea of *Two-opt* local search is iteratively taking two edges and computing if a swap of these edges improves the fitness of the individual. If it does improve the fitness, set it as new best and continue. It does this until the value does not improve anymore or it has reached a maximum of iterations. The working of *nearest neighbor* was already explained in section 4.4. These two operators were one of the most important additions to the algorithm, since they have a big impact on the performance and fitness of all the individuals. They improve the fitness of each individual in such a way that the whole algorithm is sped up.

I started with a very basic local search operator, similar to the one in the professors' tutorial on local search operators. But since the *two-opt* search outperformed it, it was discarded. The parameters for *nearest neighbor* are already discussed in the initialization section. For *two-opt*, the parameter that needed to be tuned is the maximal amount of repetitions that improve the route.

4.10 Diversity promotion mechanisms

To promote diversity in the algorithm, the population is divided into two disjoint subpopulations. After initializing the population, it is split up into two **subpopulations** or islands. This approach is called **the island model**. The two islands are optimized independently. The main difference is the *recombination operator*. One uses SCX recombination and the other uses order crossover recombination to create two diverse populations. After a fixed amount of iterations, **individuals are exchanged** between the islands. This exchange is done with k-tournament selection and the same value for the parameter k as in selection and elimination. If one of the islands converges, the two islands are **merged to a new population** and the algorithm continues with *order crossover recombination* until the global convergence criteria are met.

Since these two recombination operators have different properties and outcomes, the two islands differ in their evolution. They both produce good solutions, but these solutions are often not alike. When exchanging these individuals, a divergent but good solution might **help an island from getting stuck in a local minimum**.

The parameters that need to be determined are the amount of iterations before an exchange and the amount of individuals exchanged. The former is set to 10 percent of the total amount of iterations the algorithm can execute. The latter is set to 10 percent of the amount of individuals per island.

4.11 Stopping criterion

I combined several criteria to determine when the algorithm needs to stop. This essentially can be split up into two main motivations: **time/duration** and **convergence**.

The first is determined by the time left and directly determines the maximum amount of iterations. Since the

assignment states that the algorithm only has five minutes and a restriction of two cores, it is hard to exactly mock the circumstances it will run in during the tests. This is why the algorithm constantly adds the timings of every iteration to a list and computes the average. Based on the total given time and the average time per iteration, it is straightforward to set a maximum amount of iterations with the formula below. This maximum amount has a big **margin for unexpected delays**.

$$max\_amt\_its = \lfloor \frac{max\_time}{avg\_its\_time} - 10 \rfloor \tag{6}$$

The latter is determined by the consecutive amount of iterations with the same best value. When the algorithm return the same value as best value for 100 times, the algorithm stops.

4.12 Parameter selection

To tune the parameters that are not automatically determined with adaptivity, I used **hyperparameter search** (grid search) and trail-and-error. The hyper parameter search was executed by repeatedly running the algorithm with different values for the parameter on the *tour250* problem. This problem, containing 250 cities, is big enough to create an estimate of the impact.

k (k-tournament): The value of k influences the bias of the algorithm towards **exploration or exploitation**. To determine this value, the algorithm was executed with values ranging from 1 to 5. Figure 2 shows the result of this search. A **value of 3** for k gives the best result out of the tested options.

μ: The size of the offspring was found by performing a grid search on the **ratio** $\frac{\mu}{\lambda}$. Since computational complexity plays an important role in this decision, figure 2 shows the amount of iterations on the x-axis and the best fitness on the y-axis. It clearly proofs that a **value of 1** is the best choice. A smaller offspring means less computations and therefore more iterations within the time limit. Being able to do more iterations significantly improves the best value.

$p_{nearest\_neighbor}$: This value determines the probability of an individual undergoing the nearest neighbor optimization in the initialization phase. After experimenting with this value using trial-and-error, a hyper parameter search was performed with values 0.01, 0.05, 0.1, 0.2 and 0.5. The result can also be found in figure 2. It shows that both the lowest and highest value perform the best. Since the highest value has a big impact computationally, the lowest value, **0.01**, was chosen.

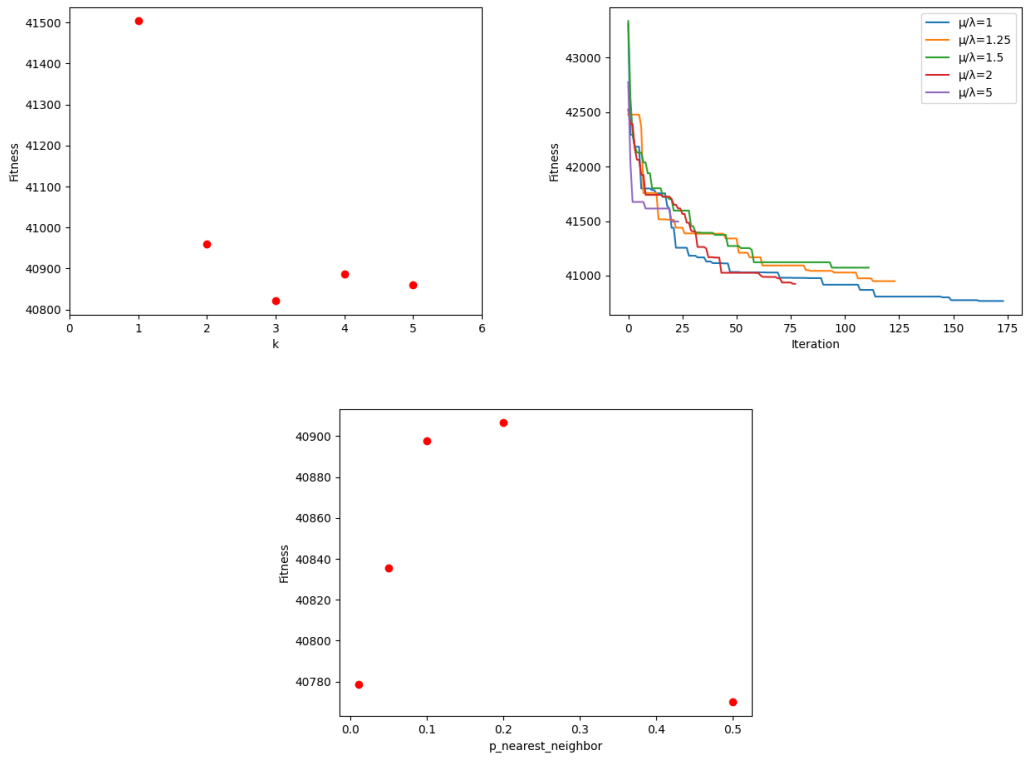

Figure 2: Hyperparameter search for k, μ and $p_{nearest\_neighbor}$

$p_{recombination}$: To introduce more variety in the algorithm, the **probability of recombination is 1**, which means

that every individual in the offspring is the child of two parents.

$p_{two-opt\_lso}$: The powerful two-opt local search delivers one of the most important contributions to the performance of the algorithm. With the help of the JIT-compiler, it is computationally possible to execute this local search with a **probability of 1**.

$max\_its\_same\_best$: The maximum allowed amount of iterations with the same best value is **hardcoded to 100**. The reason for this static value is that, given the time limit, the exact amount of steady iterations does not matter as long as it is high enough. For smaller problems, it is better to make sure that the algorithm converged and to do some redundant iterations than to quit to early.

$ile\_its\_before\_exchange$: **Dividing the total amount of iterations by 10** gives an acceptable value for the amount of iterations before each exchange between islands. This value usually lies between 10 or 25 iterations, which is a common value for island iterations according to the lecture slides and the algorithms performance.

ile_swaps: The amount of individuals swapped in every exchange is set to **5 percent of the total population**. This means that 10 percent of every island is swapped with 10 percent of the other island.

Parameter	**Value(s)**
k	3
μ	λ
$p_{nearest\_neighbor}$	0.05
$p_{recombination}$	1
$p_{two-opt\_lso}$	1
max_its_same_best	100
ile_its_before_exchange	$\lfloor \frac{max\_its}{10} \rfloor$
ile_swaps	$\lfloor \frac{\lambda}{20} \rfloor$

Table 1: Parameter Selection

4.13 Other considerations

Next to the already implemented components, two other items were considered. The first one is adapting the island model to a multi-objective setting and the second one is penalizing items individuals close to each other in elimination. Both ideas are not elaborated since I wanted to put more effort in optimizing the current algorithm setup, than in adding as much strategies as possible.

5 Numerical experiments

5.1 Metadata

The following **static parameters** are used to run the numerical experiments on the different problems.

- k: 3
- $p_{nearest\_neighbor}$: 0.05
- $p_{recombination}$: 1
- $p_{two-opt\_lso}$: 1
- $max\_its\_same\_best$: 100

Tour	$\lambda = \mu$	island swaps
tour29	180	9
tour100	180	9
tour500	120	6
tour1000	60	3

The **main characteristics of the computer** on which the algorithm ran are listed below.

Characteristics	
Operating System	Linux Mint 20 Cinnamon
Processor	Intel Core i7-4790 CPU@3.60GHz*4
Memory	16GiB
Python Version	3.8.10

Table 2: Computer Characteristics

5.2 tour29.csv

The best tour length found is **27154.48839924464**. The corresponding sequence in this case is the following:

```
[14 11 10  9  5  0  1  4  7  3  2  6  8 12 13 15 23 24 26 19 25 27 28 22 21 20 16 17 18]
```

The algorithm converges on average after just three iterations or 2 seconds. My solution beats the given greedy heuristic value (which is 27200), which means it is close or equal to the optimum.

After running the problem 1000 times and recording the results, these plots were created. The left plot shows the convergence. Since the global minimum is quickly found and the search space is small, diversity is rather low, as can be seen by the mean fitness in the plot.

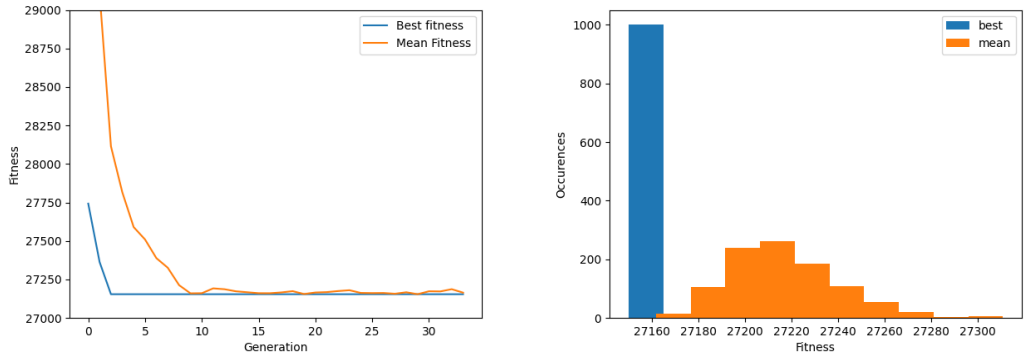

Figure 3: Convergence plot (left) and histogram after 1000 runs (right) of the mean and best fitnesses of tour29

The orange histogram displays the final mean fitnesses. The mean value of all the **mean fitnesses is 27215.886472712762** and the **standard deviation is 22.28166719746619**. The mean value of the final best fitnesses is equal to the best tour length, because the algorithm always finds the optimal value. Therefore the standard deviation is zero, there is no spread.

5.3 tour100.csv

The best tour length found is **221568.56127564854** after **170 iterations and 34.52 seconds**.

The amount of cities in this tour is smaller, so the average time per iteration is much lower than with bigger problems. On average it takes about from 200 to 250 iterations and from 50 to 60 seconds for the algorithm to converge. In the first few iterations the objective value goes down because of the local search operators. In the further iterations, it tries to find a better solution by mutating the new offspring.

5.4 tour500.csv

The best tour length found for **500 cities** is **101765.50862028853 after 74 iterations**. On average the final best fitness value is around 102500 in 70 to 80 iterations and the algorithm takes 270 seconds.

The algorithm converges to local minima in the first 40-50 iterations. Because of the diverse population and the merger of the two islands, it manages to get out of those local minima and optimize even further.

5.5 tour1000.csv

The best tour length found is **184721.5718169272** after **158 iterations in 280 seconds**.

For the largest possible amount of cities, the algorithm takes nearly the full five minutes to converge. Since the algorithm is designed to converge within this time limit, it performs well. The diversity is sufficient to keep

lowering the best objective value. In most of the cases, it would probably find an even better solution without time constraint.

The simple greedy heuristic for this tour size is 226541. With a best tour length of 184721 and an average around 185500 it easily outperforms this value.

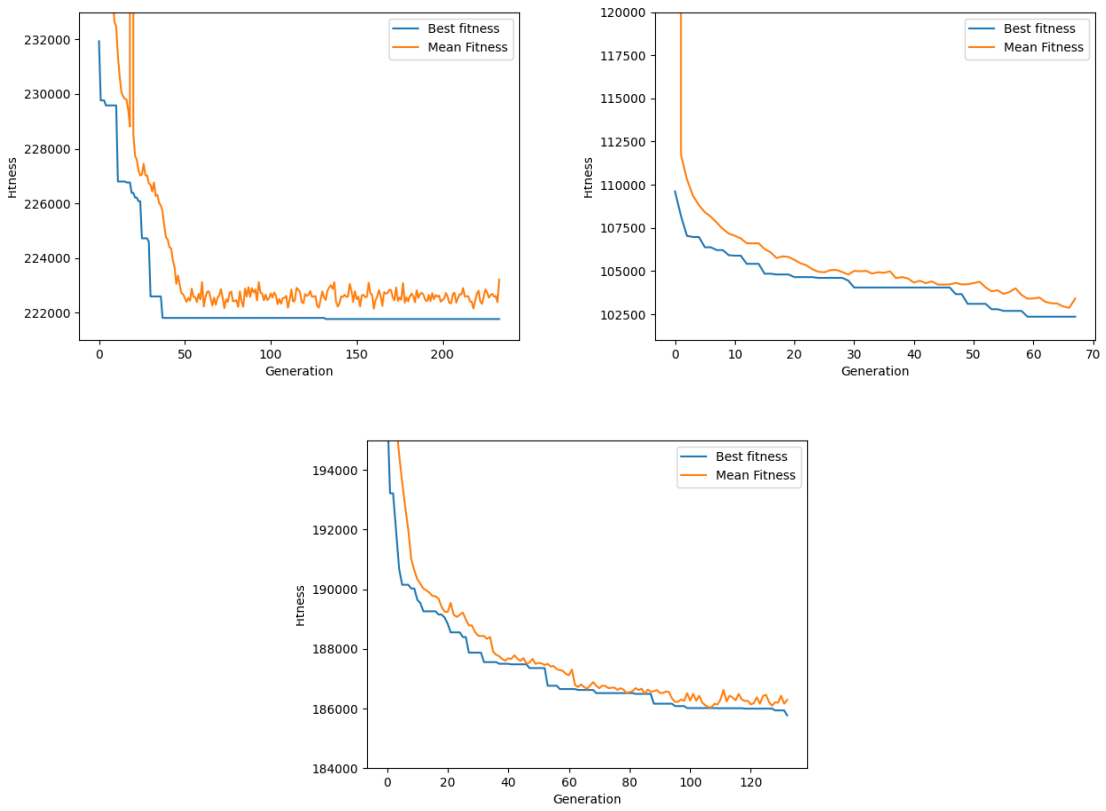

Figure 4: The plot of mean and best objective values of tour100, tour500, tour1000 respectively

6 Critical reflection

During the lectures I learned about the **theoretical aspects** of an evolutionary algorithm. The concepts of different operators, components or techniques were explained and pretty clear. However, the only way to really get a grasp of the working was implementing my own algorithm. Sketching an idea of what elements it should contain is straightforward, but **analyzing problems, finding weaknesses or getting the different operators work together efficiently** in the algorithm are the most difficult, yet most useful part of working on the assignment. I definitely believe an evolutionary algorithm is appropriate for the Traveling Salesman problem. There is a clear way to define fitness and the search space is well defined.

I learned how to make an efficient evolutionary algorithm wherein the different aspects work efficiently together to find the best value, while balancing between exploration and exploitation. The thing that surprised me the most is the time invested in writing an operator in comparison to the **time invested in making the operator work efficiently** in the algorithm. Writing an operator is relatively easy, but making it work in an algorithm is a complicated process.

6.1 Main strengths

1. The **strong initialization** gives the evolutionary algorithm a **head start** in finding the best solution. By applying the nearest neighbor algorithm to some individuals in the initialization, a slow or cold start is prevented. The individuals improved by nearest neighbor, lower the mean fitness of the population and provide some anchor points to work with.
2. Another important strength of the algorithm is the **2-opt local search**. With the usage of the JIT-compiler [7], lots of iteration can be executed to improve every individual from the offspring. If an already good offspring is created by the variation operators, this local search method makes them even better.
3. The split up of **two recombination operators on different islands** is the third main strength. Working with two islands allows the algorithm to discover solutions in two different ways. Each of the islands has

their own way of searching through the solution space and generates unique solutions sets. With a single population (1 island) it might get stuck in a local minimum, whereas with multiple islands, it gets the input from an independent other mechanism. This allows the algorithm to keep a diverse population.

6.2 Main weak points

1. For larger problems, in some cases **much time is wasted in a suboptimal region**. When the algorithm eventually finds a better region, the solution again quickly improves. With the strict time limit, this might be too late and the algorithm converges too late.
2. Although the local search methods are also in the list of three main strengths, they have one big weak point: **computational complexity**. This is especially the case for 2-opt. The complexity of this method is $O(n)$, which has a big impact on the speed of the algorithm. Lowering the speed, means a longer average time per iteration, which means less iterations in the same time. Since the improvements on the individuals due to two-opt are very good, it is worth the complexity.
3. As shown in the numerical experiments for the 1000 cities problem, the algorithm **does not converge completely in time**. One of the causes might be the first already mentioned weak point.

References

[1] R.H. MURGA I. INZA . LARRAN AGA, C.M.H. KUIJPERS and S. DIZDAREVIC. *Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators*. Kluwer Academic Publishers, 1999.

[2] Zakir H. Ahmed. *Genetic Algorithm for the Traveling Salesman Problem using Sequential Constructive Crossover Operator*.

[3] Anuj Bhai Mehta Abhishek Chelawat Dr. Urjita Thakar Akshay Vyas, Dashmeet Kaur Chawla. *Genetic Algorithm for solving the Traveling Salesman Problem using Neighbor-Based Constructive Crossover Operator*. 2018.

[4] G.A. Croes. *A Method for Solving Traveling-Salesman Problems*. 1958.

[5] L. Davis. *Applying adaptive algorithms to epistatic domains*. Morgan Kaufmann Publishers Inc., 1985.

[6] A.E. Eiben and J.E. Smith. *Introduction to evolutionary computing*. Springer, 2003.

[7] numba. numba. https://github.com/numba/numba.