Evolutionary Algorithms: Final report

Karel Everaert (r0856880)

January 29, 2021

1 Metadata

- **Group members during group phase:** Matthias Maeyens and Sander Prenen
- **Time spent on group phase:** 13 hours
- **Time spent on final code:** 35 hours
- **Time spent on final report:** 11 hours

2 Modifications since the group phase

2.1 Main improvements

Adding local search to the initialization method: In an attempt to create a better initialisation function I added local search to the initialisation function that was initially used. For more information about this improvement section 3.2 can be consulted.

Making sure that the best path doesn't get lost: In the first version of the algorithm the best path of a certain generation could undergo mutations. This had as an effect that the path that was the best in the final generation could be worse then a path that was found in an earlier generation. Section 3.6 describes how this elitism was implemented and what kind of effects this had on the quality of the algorithm.

Optimizing the mutation operator: After a hyper-parameter search on the first version of the algorithm it turned out that the best results were found when using a very low mutation rate, this can lead to a population that is not diverse enough. Instead of using another mutation algorithm the new algorithm uses two tricks to solve this. The first trick is that the top k (mostly k is 1) best solutions found in a population are not affected by mutation. The second trick is that the mutation parameter is high in the beginning but gradually goes down after the algorithm runs for a certain time. More information about this improvement can be found in section 3.4.

Optimizing the cross-over operator: The new cross-over operator follows the same principle as the one in the previous algorithm (PMX), but the length of the sub-cycle that is crossed goes down gently when the algorithm is running. Section 3.5 gives a more elaborate explanation about the exact working of the used recombination operator.

A better stopping condition: The stopping condition is of lesser importance for this specific exercise, since the algorithm will be stopped after 5 minutes. For real-life applications however it is essential that a good stopping condition is constructed. Section 3.8 gives an explanation of a stopping condition that can be used in the genetic algorithm.

2.2 Issues resolved

Mutation rate and elimination scheme: A problem with the previous version of the algorithm is that a higher mutation rate resulted in a worse result. In the group report introducing a new mutation algorithm was proposed as a possible solution. However this is not even needed since the reason for this effect appears to be the fact that the best found solution was often mutated as well and therefore lost. This ties together with another issue that was described in the group report. That issue described that the mutation of the best found solution in each population indeed causes a problem and proposes the implementation of some form of elitism as a possible mitigation strategy for this problem. I opted to implemented a form of elitism, where the number of individuals that go unchanged from one population to another grows a bit over time. In the beginning the 2 best solutions don't get changed, but when the algorithm runs for some longer this number increases a bit. This optimization improved the results significantly as can be seen in figure 2 and figure 3.

Recombination operator: Another possible issue that was identified at the end of the group phase was that the recombination operator (PMX) is designed to be used in a symmetric case of TSP. While it is true that PMX will find better solutions when this condition holds, it is not required in order to get good results. I decided to keep a modified version of PMX as the recombination algorithm. This modified version works just like the version in the previous algorithm with the single difference that the length of the sub-cycle that is exchanged between the two parents becomes smaller over time. The positive effect of this adaptation is that the algorithm easier converges in the end.

3 Final design of the evolutionary algorithm

3.1 Representation

In this implementation an individual is represented by the cyclical order of the visited locations. This cyclic order is stored in a numpy array. During the group phase we discussed the option of using a python list in order to represent an individual. This proposition was rejected since this might be less efficient. The use of a set to represent an individual was not an option since a set is an unordered collection and therefore not capable of representing an order of visited locations. To represent the entire population, we also used a numpy array of numpy arrays. We could have also used a python list, but did not do this due to the reason previously mentioned. We discussed the use of a set for this purpose, but we want to allow multiple copies of an individual in the population and a set does not allow this. In the end we sometimes had to convert the population to a list anyway for sorting, for example during the elimination step.

3.2 Initialization

The initial idea for the initialization algorithm was just to generate random individuals to initialise the population. A random individual is created by a permutation of the possible locations. We believed that this way of initializing the population should result in enough diversity. However this assumption does not hold. A randomly initialized population does not necessarily lead to a diverse population. In order to see how diverse a population is you should check the variance.

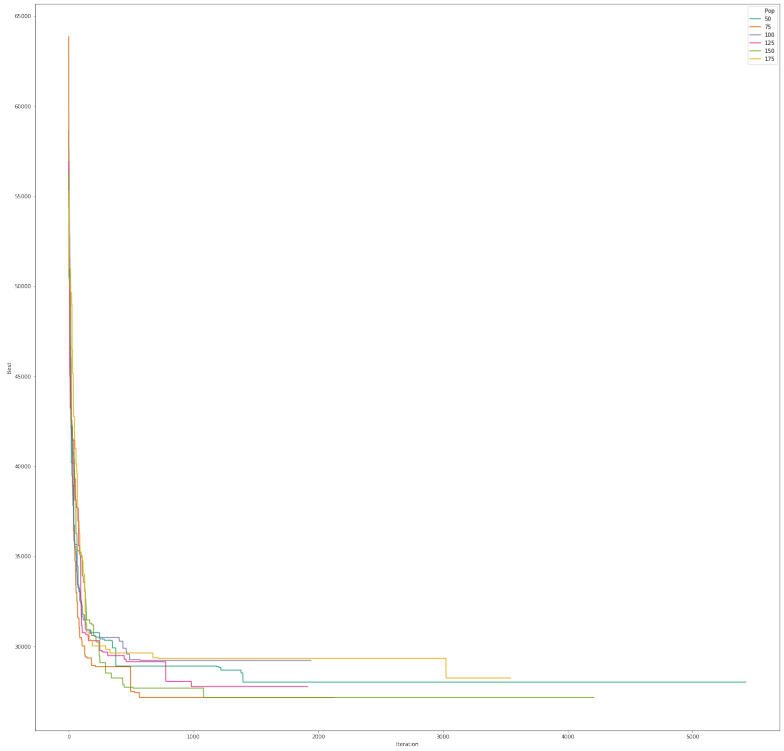
In an attempt to create a better variation on this algorithm I decided to add local search to it, under the assumption that starting with a better initial population might lead to a better solution in the end. The local search method take an individual and creates a very similar individual by just swapping 2 cities with each other. The use of this local search method introduced some additional hyperparameters, the first one being alpha which is a float between 0 and 1.0 that determines the chance that local search is performed on a certain individual in the population. The hyperparameter k determines the number of similar instances will be generated when local search is performed. While depth represents the amount of time local search will be performed e.g. when depth is 2 then local search will also be performed on the randomly generated similar instance. The working of this initialization algorithm is as follows: a random individual is generated and based on a random value and the value of alpha is determined if local search is performed on this instance. If local search is performed then k variations of the original individual are generated and when an individual turns out to be better then the original one then it replaces the original one. This procedure is repeated depth times. The use of local search didn't have a lot of impact on the solution that was found or the time it takes to reach this solution in smaller graphs. The reason for this can be seen in figure 3. This graph displays the length of the found path on the Y-axis and the iteration on the X-axis. The sharp drop on the left of the graph immediately draws attention and explains why local search doesn't have a lot of impact. This sharp drop indicates that the algorithm almost immediately drops from a best length of 65000 to a length of 30000, so whether you start from a best length of 70000 without using local search of from 65000 with the use of local search will almost have no impact.The use of local search is better in larger graphs where the sharp drop at the start is not present.

During the analysis of the effect of the population size during the group phase we came to the conclusion that a smaller population size leads to better results.The reason behind this remarkable effect was not determined. This effect however does not occur anymore, probably due to changes in other parts of the algorithm. In order to find a good population size The algorithm was run with different sizes on the sample problems. The optimal size seemed to converge in all the sample problems to a value of approximately 75 individuals. This value seemed to allow the algorithm to consistently find a good path while not being too large that it became computationally too expensive.

3.3 Selection operators

We considered either k-tournament selection or a ranking based selection. In the end we chose k-tournament selection because it is very computationally efficient and easy to implement once we had a proper fitness function. The only parameter that needs to be picked for this is k. This determines the size of the initial randomly selected

Figure 1: Visualisation of reason why local search during initialization has not a lot of impact



group. During the hyperparameter search it turned out that a value of 10 for k is a good trade-off between speed and the quality of the individuals that are selected.

3.4 Mutation operators

For the mutation algorithm we chose a simple swap algorithm that swaps the position of two randomly selected subsets. With a parameter α that controls the chance that the subset size will increase, with a starting subset size of 0 (no mutation). This should increase the randomness sufficiently depending on the size of alpha. With the smaller alpha the less chance for a mutation. So each iteration a random number between 0 and 1 is selected. If it is lower than α the subset size is increased by 1. This is done until the randomly selected number is higher. After which two subsets of the selected length are swapped.

The mutation rate, α, that is used in the algorithm is not constant but decreases over time. The reason for this is that the algorithm needs to converge in the end and therefore needs less randomness while in the beginning the algorithm needs to discover a lot of possibilities and therefore the amount of randomness in the algorithm needs to be higher.

Another addition to the mutation algorithm that was made is that local search is performed after the mutation operator was performed on a population. This resulted on average in a better final solution.

3.5 Recombination operators

Our algorithm currently uses partially mapped crossover (PMX) as recombination operator. This algorithm picks two points at random in the representations of the parents. These two points are referred to as the cut-points. In order to generate the children the nodes between these two cut-points get swapped between the two parents. After this mapping the children often contain a certain node two times. This is not allowed in the representation, since this does not represent a possible candidate solution, and has to be fixed. In order to fix this errors the algorithm keeps track of which nodes of parent1 were mapped on which nodes of parent2. The next step of the algorithm uses this mapping to give the nodes that were not situated between the two cut-points a new value if this would be needed. If in a child a node outside of the two cut-points has a value that is now present between between these two points than the mapping provides you directly or indirectly with the new value that this node should receive. Other recombination operators were not considered during the group phase because the textbook of Eiben and Smith mentioned the suitability of PMX for the TSP. One of the possible points of critic you can give on this algorithm is that it will work best when the distance from e.g. point A to point B is similar to the distance from point B to point A. While this is true I assume PMX is still a good choice for the travelling salesman problem even when this condition is not satisfied since it consistently delivers good results when it is used in the algorithm.

After the group phase I decided to do only a small modification to the PMX algorithm since it already gave good results. The modified version of PMX that is used at the moment has a limit on the maximum length that can be between the two cut points. This maximum length decreases over time. The reason for this is that it might be beneficial to only perform small changes to the individuals at the end of the algorithm.

3.6 Elimination operators

We think that $\lambda + \mu$ -elimination is a suitable elimination scheme for this project. This guarantees that the best individuals in the population and offspring will survive. However, due to mutation it is possible that some of the best individuals in the original population will be altered. This may cause the best individual to be suddenly one of the worst individuals. Therefore it may be useful to always keep the best individual of the previous generation, in order to not worsen the best objective value. For this reason elitism was implemented. The parameter k determines how many individuals go unchanged from the current population to the next. This parameter starts with a value of 1 since in the initial phase of the algorithm a lot of discovery has to be done. The parameter gradually increases a bit over time, but will always stay relatively small to the population size. This modification had a huge positive effect of the quality of the best solution that was found by the algorithm. In figure 2 it is clear that the length of the best path starts to fluctuate due to the fact that the best solution can become worse, this is not the case in figure 3 since the best found solution can only become better over each generation.

Figure 2: Fluctuation of the length of the best path

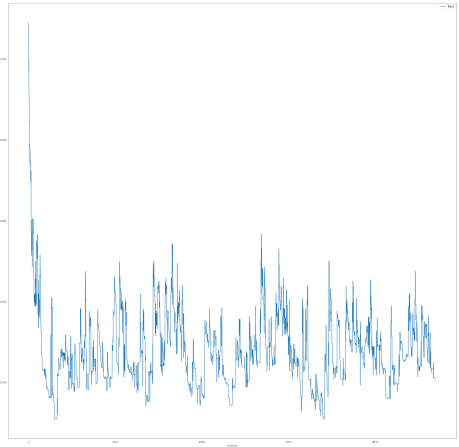
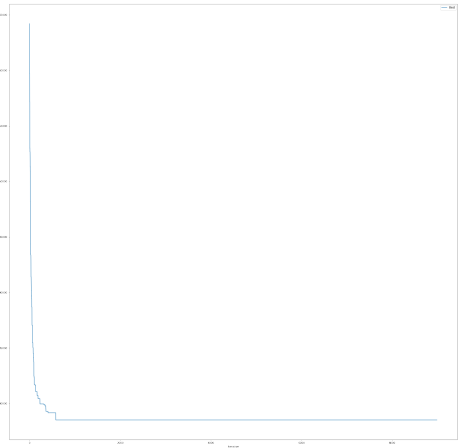


Figure 3: Stabilisation of the length of the best path



3.7 Local search operators

Local search operators as described in section 3.2 where added to the initialization and mutation step. Currently the local search operator is not longer performed during the initialization step since, it doesn't really improve the algorithm.

3.8 Stopping criterion

The evolutionary algorithm should stop when the population stays the same over many iterations and/or when the best values of the population don't change after a certain amount of iterations. Via some trial and error we created a simple stopping algorithm that stops when the best value hasn't changed in 50 iterations.

A second version of the stopping algorithm also looked at the mean value of the population, since for example if the best value doesn't change for 50 iterations but the mean value is still increasing the algorithm should not stop looking for a better solution.

3.9 The main loop

Figure 4: Main method of the genetic algorithm

```
def optimize():
    population = initialize()

    while (stoppingCriteriaNotSatisfied() and iterationsLeft()):

        for all pairOffParents
            parent1 = selection()
            parent2 = selection()
            child1,child2 = PMX()
            offspring.add(child1,child2)

        for j in range(len(offspring))
            mutated = mutation(offspring[j], chance)
            if performLocalSearch
                mutated = localSearch(mutated)
            offspring[j] = mutated

        sorted(population, key=lambda individual: length(individual, distanceMatrix))
        for j in range(elitismOffset, len(population))
            mutated = mutation(population[j])
            if performLocalSearch
                mutated = localSearch(mutated)
            population[j] = mutated

        population = elimination(population)

        timeLeft = reporter.report()
        if timeLeft < 0:
            break
        i+=1

    return 0
```

The pseudo-code of the main method used in the genetic algorithm can be found in figure 4. The order of the various operations is as follows: first the start population gets initialized, this is followed by generating the offspring by performing the PMX-recombination operator. The mutation operator gets performed with a certain chance over the entire offspring. For the parent population elitism prevents that the 'elitismOffset' best individuals don't get mutated but the rest of the parent population can get muted with a certain chance. After this step the elimination step is performed to make sure that the next population only contains 'populationSize' individuals. Due to the fact that all this operations are performed in a while loop the order can be changed, however this seemed to be the most logical way to structure the algorithm in my opinion.

3.10 Parameter selection

In order to determine the value of the important parameters that are being used in the algorithm a hyper-parameter search was performed where a lot of combinations with different values for the important parameters where tried. Out of this combination different good combinations where determined for the different sizes of the problems.

4 Numerical experiments

4.1 Metadata

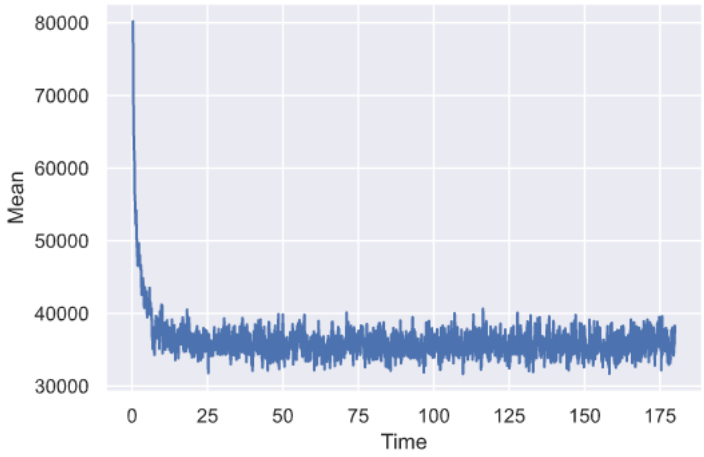
There is a large number of parameters that can be set to change the behavior of the genetic algorithm. A list of the most important parameters, together with their meaning and eventually value, can be found here:

- population_size: Determines the size of the population. The value of this parameter should depend on the scale of the network it is run on. Due to a shortage of time on my side I was only able to do a parameter analysis on the networks of 29, 100 and 194. Out of this data I concluded that a population size of 75 individuals is a good choice.
- its: The its-parameter can be used to set an upper-bound to the number of iterations that the algorithm can run. This setting is not important anymore, but was very useful during testing and debugging.
- recom_its: This parameter determines how much recombinations are performed. This parameter was set default to 50. A possible improvement to the algorithm is to make the value of this parameter dependant on the size of the network.
- k: Parameter k is used in k tournament selection, where out of k random individuals the best one is selected. This parameter should depend on the size of the network as well, but for now it is set to 10.
- alpha: Alpha is used in the mutation operator and determines the chance that an individual gets mutated.
- chance_local_search: Determines the probability that local search is performed on an individual. This parameter is by default set to 0.5.
- startIndex: This parameter is used to implement elitism. The 'startIndex' best individuals of a population will go over to the next population with a probability 0 of being mutated. The interesting aspect of this parameter is that, since it is known that the algorithm will run for 5 minutes, the value of this parameter has been made dependant on the running time. In the beginning only a small amount of individuals will get included in elitism since this is good for exploration. Later a bit more individuals are included.

The evolutionary algorithm was run on a laptop with an Intel core i7 CPU, with 2 cores and a clock speed of 2.40 GHz and a memory of 16 GB. The version of python that was running on the laptop was version 3.8.5.

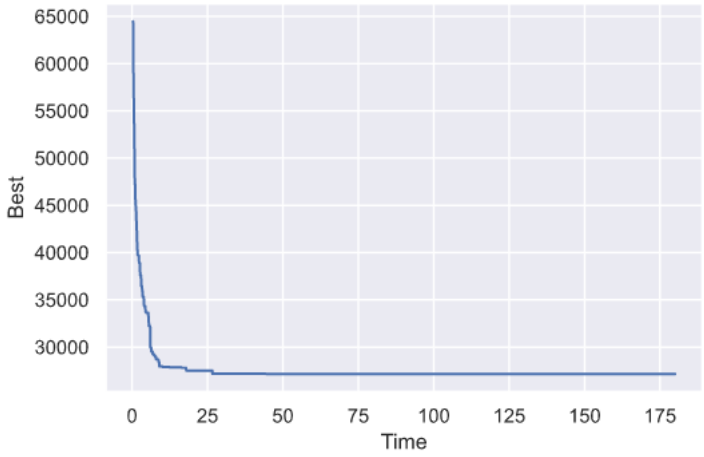
4.2 tour29.csv

Figure 5: Change of the average path length of the population over time in tour29



A typical convergence graph for the mean and best objective values in function of the time can be found in Figure 5 and 6. The graphs show that it doesn't take long before a good solution is found. The best path that was found had a length of 27154.488399244638. The sequence of cities that corresponds with this best length is (23, 24, 11, 9, 2, 7, 14, 22, 13, 18, 19, 27, 21, 26, 15, 16, 20, 25, 10, 5, 1, 0, 4, 8, 12, 6, 3, 17, 28). Since the optimal value

Figure 6: Change of the length of the best path of the population over time in tour29

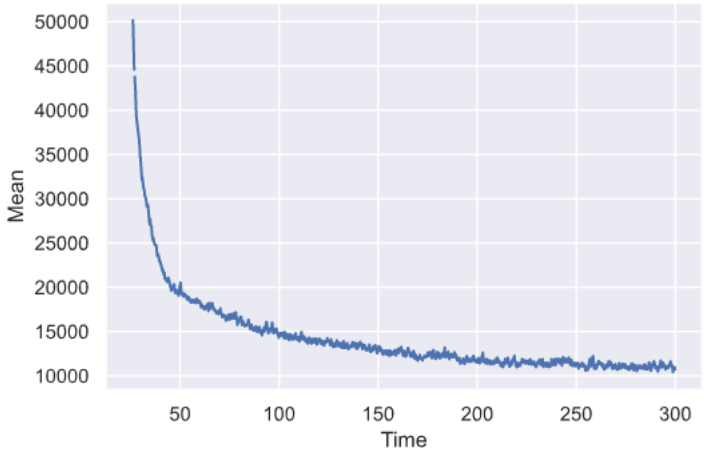


is approximately 27200 I conclude that the algorithm is sufficiently powerful to find a good solution in small networks.

4.3 tour100.csv

The best solution that was achieved had a length of 9141.544546895373 which I think is acceptable. When testing the algorithm on this problem I noticed that the length of the final best solution varies a lot. The faster the algorithm found a path that had a length that is not 'inf' the better the final solution.

Figure 7: Change of the average path length of the population over time in tour100



In the graphs displayed in figures 7 and 8 it is clear that the algorithm converges to a good solution. It is interesting to see that there is a gap in the graph in figure 7.

4.4 tour194.csv

When looking at the 2 graphs in figures 9 and 10 it becomes clear that it is a lot harder to find a good path then it was in tour29. However the algorithm still manages to converge to a good solution. The best tour length that was found was 13026.94852972119. Since the optimal solution approximately is 9000 the algorithm is not yet optimally for networks of this size. A possible way on how I would improve the algorithm is to do an extensive hyper-parameter search on this network.

4.5 tour929.csv

In the graphs in figures 11 and 12 you can see that the algorithm is on the right track but did not yet have the time to converge. The current algorithm is currently not powerful enough to handle networks of this size in the restricting 5 minutes. The algorithm did 275 iterations before the timer stopped.

Figure 8: Change of the length of the best path of the population over time in tour194

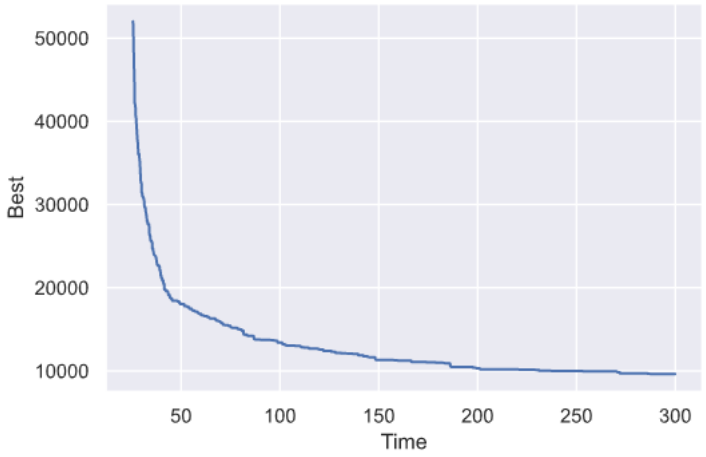
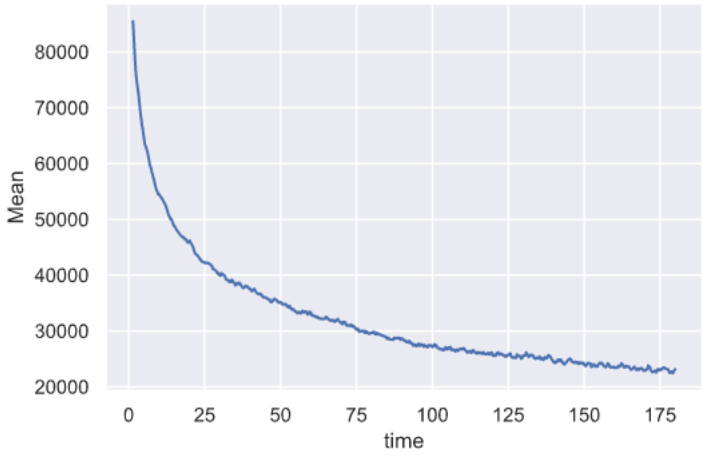


Figure 9: Change of the average path length of the population over time in tour194



5 Critical reflection

During the course of this projects I learned a lot of practical insight in how to construct an evolutionary algorithm from scratch. I think one of the main strong point of the use of genetic algorithms is that it is very easy to get some results once you have a way to describe your problem to a computer it is easy to construct a very basic evolutionary algorithm. The downside is that there are a lot of parameters to tune and due to the randomness it can be hard to figure out how to make the algorithm perform better. I don't think this downsides can be avoided, but the next time I construct an algorithm like this I will focus more on finding a better overview of all the parameters within the code. Personally I think the most useful thing I learned during this project is to look up a good method in a paper and being able to actually implement this method e.g. PMX.

Figure 10: Change of the length of the best path of the population over time in tour194

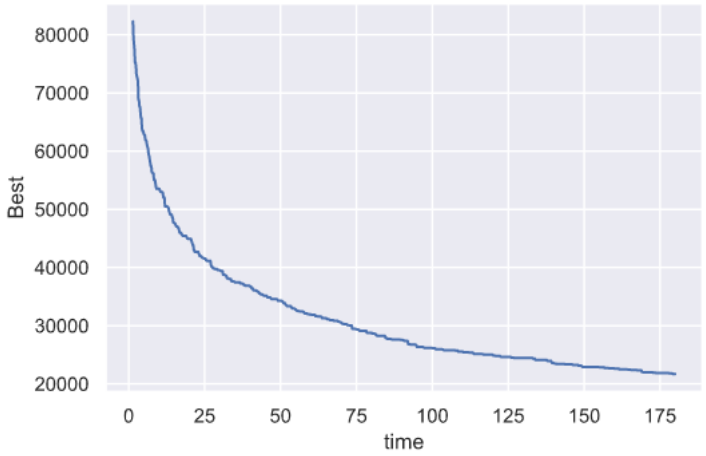


Figure 11: Change of the average path length of the population over time in tour929

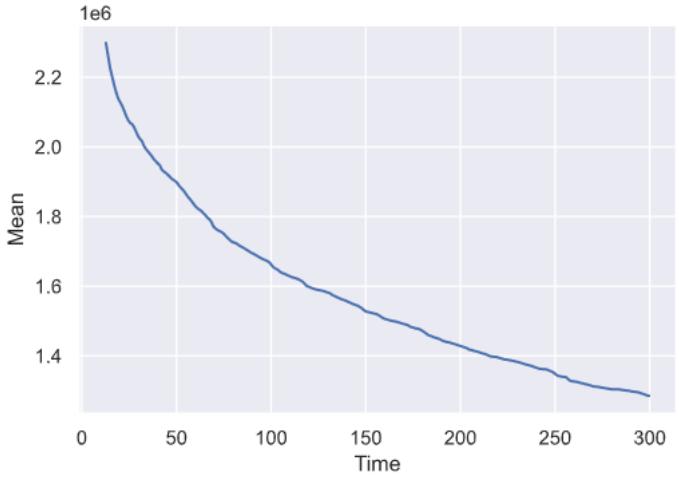


Figure 12: Change of the length of the best path of the population over time in tour929