Evolutionary Algorithms: Final report

Simon Perneel

December 30, 2020

1 Metadata

- **Group members during group phase:** Thomas Feys & Timo Martens
- **Time spent on group phase:** 10 hours
- **Time spent on final code:** 42 hours
- **Time spent on final report:** 12 hours

2 Modifications since the group phase

2.1 Main improvements

The following paragraphs briefly describe the main improvements since the group phase. A more detailed explanation is given in section 3.

Initialization of the population: The population is initialized now with an amount of heuristic good individuals. It leads the search into more promising areas of the search space and speeds up the optimization process. See section 3.2 and 3.7.

More and better mutation operators: In addition to swap mutation, insert mutation, scramble mutation and reverse sequence mutation has been implemented in order to increase randomness in the search. It results in more diversity in the population. See section 3.4.

Dpx crossover and cycle crossover: In addition to order mutation, distance preserving crossover and cycle crossover have been implemented in order to generate more diverse individuals. See section 3.5

Crowding: The (λ+μ)-elimination has been modified with a crowding algorithm to promote diversity and avoid premature convergence. See section 3.8.

Elitism: Elitism is used to ensure that the best individual of the population survive elimination and remain unmutated. See section 3.12

Code optimizations: Some improvements have been made to the code in order to reduce iteration time and obtain more overview in the code. This is explained more in detail at section 3.12.

2.2 Issues resolved

Handling problems with disconnected cities: With the first version, the algorithm did not work with cities that are not connected with each other. The algorithm could not deal with the infinite costs in the distance matrix. This problem is solved simply but effectively by replacing the infinite values in the matrix by extremely high costs. In this way, tours with unconnected cities are not selected to generate offspring and will rapidly be eliminated from the population.

Early convergence: The diversity of the population in the first version of the algorithm decreased too quickly. This led to an early convergence of the mean fitness and best fitness. Because of this, the algorithm often gets stuck in a local optimum. This early convergence was mainly due to the (λ+μ)-elimination scheme and the order crossover operator. These operators cause a loss in diversity. In addition, there was no diversity promotion implemented yet.

This problem was solved by using crowding as a diversity promotion scheme. Whenever an individual is promoted to the next generation, the algorithm removes a similar individual in the seed population. This is further explained in section 3.8. Also, the distance preserving crossover (DPX) has been implemented. This crossover

operator makes use of instance-specific knowledge, while at the same time, it preserves diversity within the population. See 3.5 for a more detailed explanation.

Not optimized for larger tours: During the group phase the code was too slow for the larger tours. The algorithm ran out of time, with the optimal and heuristic value still quite far away. The problem was that the algorithm converges too quickly and easily gets stuck in local optima. The iterations for calculating new generations also took too long.

This problem has been addressed by several techniques. By use of a heuristic to initialize the population, the algorithm could start the search at a more promising area in the search space. Therefore, the algorithm yielded better results. The use of the heuristic to initialize the population creates a new problem; this field in the search space has a lot of local optima. Therefore, the alpha value has been set quite high from the beginning. The code has also been optimized which such that the iterations are done faster.

Now, the found result is always better than the heuristic value. However, for the largest tour, the result is still too far away from the optimal value after 5 minutes.

3 Final design of the evolutionary algorithm

3.1 Representation

The individuals are represented by a permutation of integers. The permutation represents the order in which the cities are visited and thus the tour. The permutation appears as a numpy array of integers in the code. This representation is intuïtive and easy to work with. It allows the use of common crossover and mutation operators. In addition to the tour, one individual has some additional values that are important in the code:

- cost of the tour: used as fitness value
- edgeset: set with all the edges of the tour
- p_{rc}: probability of recombination
- p_{cw}: probability of crowding when an individual is promoted
- cost_uptodate and edges_uptodate: see 3.12

3.2 Initialization

The population is largely initialized with random individuals (i.e. random permutations) in order to start with sufficient diversity. A smaller part of the population is initialized with heuristic good individuals. The tours of these individuals are made by use of the nearest neighbour algorithm. A tour starts with a random city, and repeatedly visits the nearest city until all have been visited. This draws the search from the beginning towards more promising solutions. To prevent these heuristic individuals from immediately taking over the entire population and removing all diversity, the mutation probability has been set high from the beginning.

3.3 Selection operators

In order to generate offspring, a selection operator is needed to pick two parents from the population. K-tournament selection is used for this. It is simple but effective and has usually a low computational cost. The parameter k has been set to five. This value appears to be a good trade-off between selection pressure and computation time.

3.4 Mutation operators

Four mutation operators are implemented in order to increase randomness in the search. After all, one operator can move a tour throughout the search space in a way that another cannot. The mutation is applied to the offspring and seed population (except for the best tree individuals). The different mutation operators are listed below:

- **swap mutation:** randomly swaps two cities of the tour
- **reverse sequence mutation:** reverses the order of a random selected part of the tour
- **insert mutation:** picks a city out of the tour and insert it again at a random place
- **scramble mutation:** shuffles the cities of a random selected part of the tour

One of these operators is picked weighted. Reverse sequence mutation and scramble mutation occurs more (4x and 2x) than the other operators because they yield better solutions [1]. The probability of mutation is set with the parameter α. The α-value is set high to ensure that the algorithm does not get stuck in one of the many local

optima introduced by the initialization. In addition, the value starts at 0.3 and increases in time to the maximum value of 0.5 in order to avoid premature convergence.

3.5 Recombination operators

Tree recombination operators have been implemented: order crossover, cycle crossover and distance preserving (DPX) crossover. These are described more in detail below. In the final version, only a combination of order crossover and DPX crossover are used. The cycle crossover is discarded because the computation time was a two times higher than order crossover while it did not seem to produce better solutions. DPX crossover also has a high computational effort, but it preserves diversity in the population. For this reason, 1/10 times dpx crossover is used, and 9/10 times order crossover is used. The probability of recombination determines the chance that two parents generate a child and can be controlled. It is fixed at 0.99 as I did not find a good dynamic or self-adaptive parameter that resulted in better solutions.

Order crossover: This operator starts by copying a random segment from the first parent into the child. The remaining unused numbers are copied in the into the first child in the order that they appear in the second parent. This operator keeps the relative order of the cities, even when there is no or little overlap between the parents.

Cycle crossover: This operator starts by breaking the tours into cycles. The offspring are created by selecting alternate cycles from each parent. The procedure is shown in Figure 1. This operator preserves as much information about the absolute position in which the cities occur. This may be the reason why it does not create good new individuals for the TSP. The absolute position of a city in a tour is not unimportant, unlike the order the cities are visited.

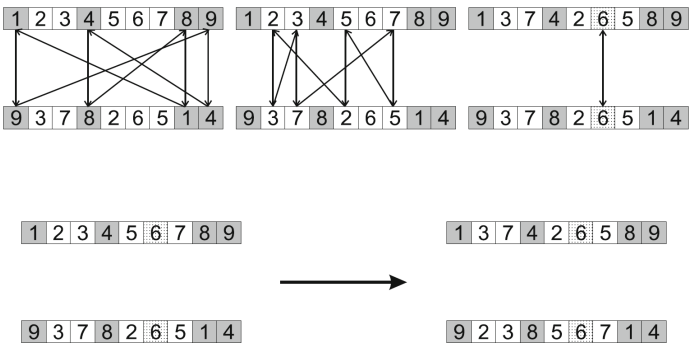


Figure 1: Cycle crossover [2]

DPX crossover: This operator has been developed for the TSP in particular [3]. This type of crossover keeps diversity by ensuring that the offspring inherits all edges common to both parents, but none of the edges that are only in one parent. This way, the generated offspring is at the same distance as its parents, hence the name distance preserving crossover. It makes use of a heuristic to join the subtours inherited from the parents. Figure 2 shows the procedure. The tour of the first parent is copied to the offspring. After this, all edges that are not common in both parents are removed. This leaves a number of fragments consisting of one or more cities. The greedy heuristic now reconnects all edges. For each end city of a fragment, it looks for the start or end city of an other fragment such that the cost between those two are minimal. It connects these two cities if this edge is not already in one of the parents (to preserve same distance, cf. supra). When there is little overlap between the parents, the common edges are preserved. But the offspring will be quite different because the edges that were not common in the parents will not be used again. This is beneficial because the (good) common parts of the parents are preserved, and the generated offspring is diverse.

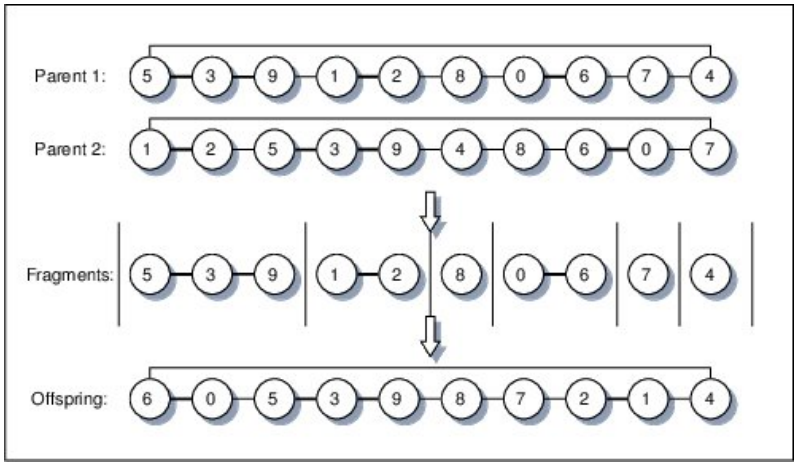


Figure 2: Illustration of the DPX crossover [3]

3.6 Elimination operators

Three elimination operators have been implemented and tested, namely (λ+μ)-elimination, (λ,μ)-elimination, and k-tournament elimination. The used scheme in the final code is (λ+μ)-elimination because it was slightly faster than k-tournament elimination. It also gave better results than (λ,μ)-elimination. Only the best individuals are promoted to the next generation, therefore it is likely to lose diversity. To counter this, a large population size of 250 individuals and crowding is used to explicitly promote diversity (See 3.8).

3.7 Local search operators

Local search has been used with the initialization of the population and with the DPX operator. A part of the population is initialized with a heuristic, namely the nearest neighbour algorithm. A tour starts with a random individual, and repeatedly visits the nearest city until all have been visited. This allows the algorithm to start immediately in a promising area of the search space and speeds up the optimization process. This is needed because of the time constraint of five minutes. Nearest neighbour also has been used in DPX crossover. This is already discussed in Section 3.5. I also considered to implement two-opt. Two-opt avoids routes that intersect each other. The basic move behind two-opt is the same as reverse sequence mutation: it selects a part of the tour and reverses the order of this part. The difference is that two-opt keeps repeating this until improvement is made. But because I already used reverse sequence mutation, it would not have a great effect. It also comes with a high computational cost and I decided to run more iterations instead.

3.8 Diversity promotion mechanisms

In order to promote diversity, crowding has been used in combination with the elimination scheme. It works as follows: it ranks the population according to their fitness as in (λ+μ)-elimination. Then the list is run through and the individual is copied each time to the next generation. Whenever this happens, a close individual is eliminated from the rest of the list. This is repeated until the population size is reached. This process is shown in Figure 3 The distance between individuals is determined by the number of common edges of their tours. All edges of an individual's tour are saved in an edge set after this, the intersection is made. A lot of common edges means a small distance between indentation and vice versa. In order to reduce computation time of crowding, the eliminated individual is determined by sampling five individuals from the population and selecting the closest one. In addition, the chance of eliminating whenever an individual is promoted (p_{cw}) has been set to 20%.

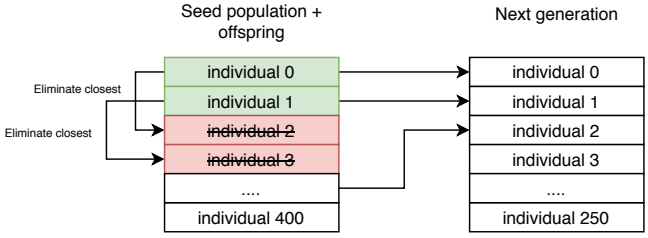


Figure 3: crowding procedure

3.9 Stopping criterion

In the final code, no stopping criterion has been used. This with the aim to find the best possible solution after five minutes. A small stop criterion has been used for testing the algorithm on tour29. Every 100 iterations, it checks whether the best individual has improved. If not, the algorithm is stopped.

3.10 The main loop

Figure 4 shows the flow of the genetic algorithm. At the start, the distance matrix is checked whether there are disconnected cities. After that, the population is initialized. 75% of the population consists of individuals with a random tour and 25% are heuristic good individuals. Now, the optimize loop starts. Parents are picked from the population with k-tournament and create offspring with either order crossover (90% chance) or DPX crossover (10% chance). When this is done, one of the mutation operators is carried out to the offspring and the population. Only the best individual of the population cannot be mutated to ensure the best solution will not change. Lastly, (λ+μ)-elimination with crowding (20% chance) is used to leave a new generation of 250 individuals. There is no universal valid approach for the order of mutation and recombination. This order is chosen because it just works. The selection/elimination for the next generation is done by selecting the ranked individuals one by one without replacement to prevent copies of individuals in the next generation. The choice of the parameters is described in Section 3.11.

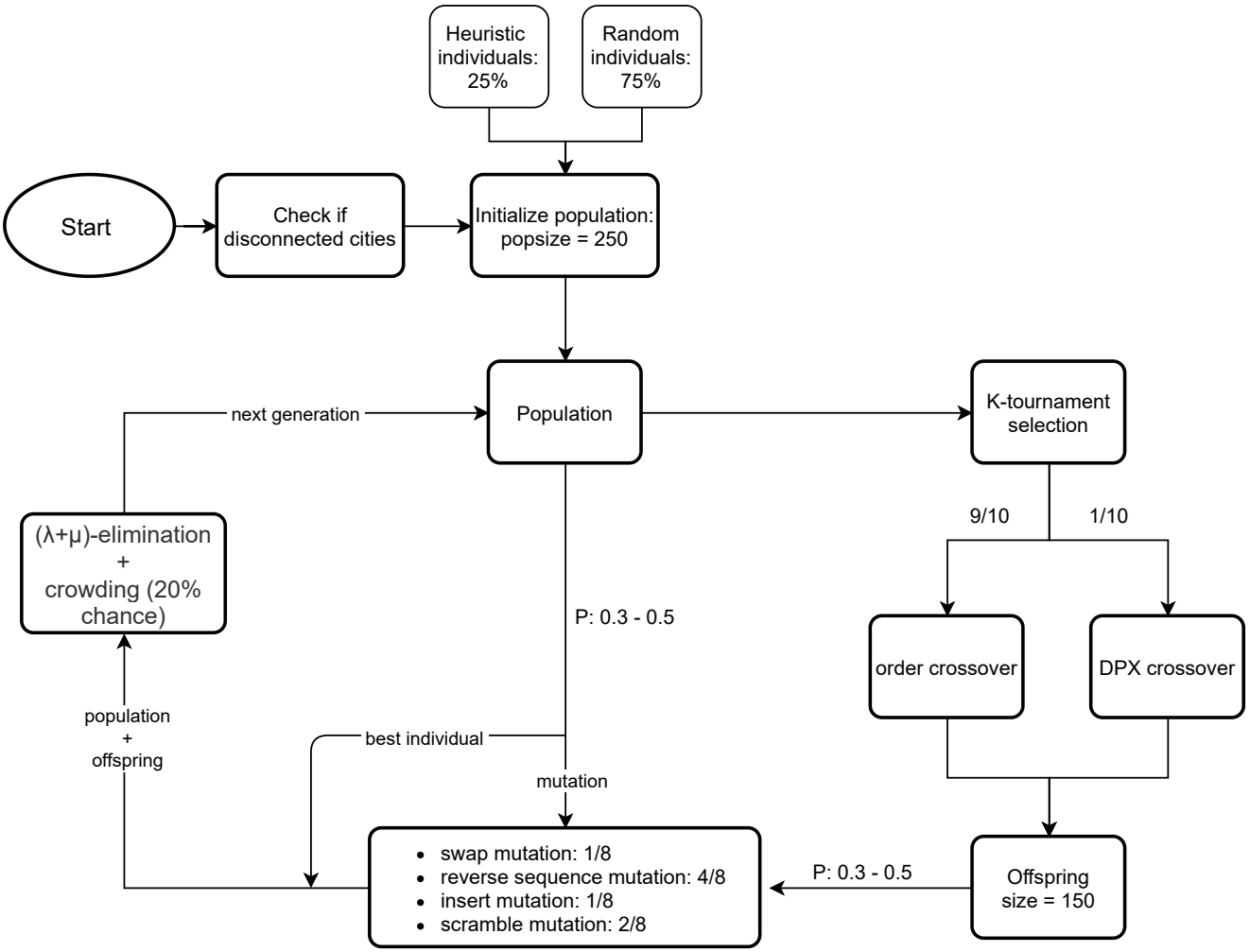


Figure 4: Main loop of the algorithm

3.11 Parameter selection

The following hyperparameters are used in the algorithm:

- population size: 250
- amount of offspring: 150
- k for k-tournament: 5

- % heuristic individuals: 25%
- probability of crowding: 15%
- probability of mutation: 30-50%

The values for the population size and k are determined in the group phase. These values turned out to give a good trade-off between their result and computation time. These values are linked because a higher population size also needs a higher selective pressure (determined by k). The other parameters are determined by trial-and-error. This is done by adjusting the parameters stepwise after 5 runs on tour194. Then the best and variation on the best individual is checked to determine the parameter. These parameter values are also valid for the smaller problems, but are not optimal on the largest tour. The larger tour may need other values in order to run faster (e.g. smaller population size).

3.12 Other considerations

Computational optimizations: Some code optimizations have been done in order to do as many iterations as possible in the five-minute time limit. Redundant calculations of the cost of a tour and the edges of a tour (for calculating distances between individuals) are avoided by saving these values in the Individual 'object' and only recalculating them when a tour has changed. This seemed a trivial thing, but actually saves a lot of time. All parameters of the GA also have been grouped in a separate class to obtain more overview in the code.

Not implemented: Some other interesting optimizations have been considered, but not implemented because of the limited time for this project. I tried implementing the island model because I believe it may yield good results. I did not proceed with this because it completely messed up the structure in my code and I no longer had a good understanding of the code.

Some things like parent selection, crossover and mutation could be done in parallel. Thus, a genetic algorithm that uses multithreading in its code could provide even more gains in computation time. I did do this because it would ask too much time and I think it falls outside the scope of this course.

4 Numerical experiments

4.1 Metadata

The parameters that were used for the evolutionary algorithm are listed below. Most of them are fixed. Only the alpha value changes over time but stays within the range of 0.3-0.5.

- population size λ: 250
- % initialized with heuristic individuals: 25%
- amount of offspring μ: 150
- k-tournament k: 5
- mutation probability α: 0.3-0.5
- recombination probability p_{rc}: 0.99

Specifications of the computer system used for the experiments:

- CPU: Intel Core i7-8550, 4 cores @1.80 GHz (code is not adapted for multi-threading)
- 8 GB main memory
- Python version 3.8

4.2 tour29.csv

The best tour length found for this problem was 27154.49. This seems to be the optimal solution for this problem. The corresponding sequence of cities of this tour is:

[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 14 11 10 9]

A typical convergence graph is shown in Figure 5. For this benchmark problem, a simple stop criterion has been used to speed up the experiment. Every 100 iterations the algorithm checks if there is any improvement. If there is not, the optimization process is stopped. The average time to find the optimal solution is around five seconds. The algorithm does perform very well for this small problem. It mostly finds the optimal solution in a short period. The diversity in the population is high enough until it finds the optimal solution.

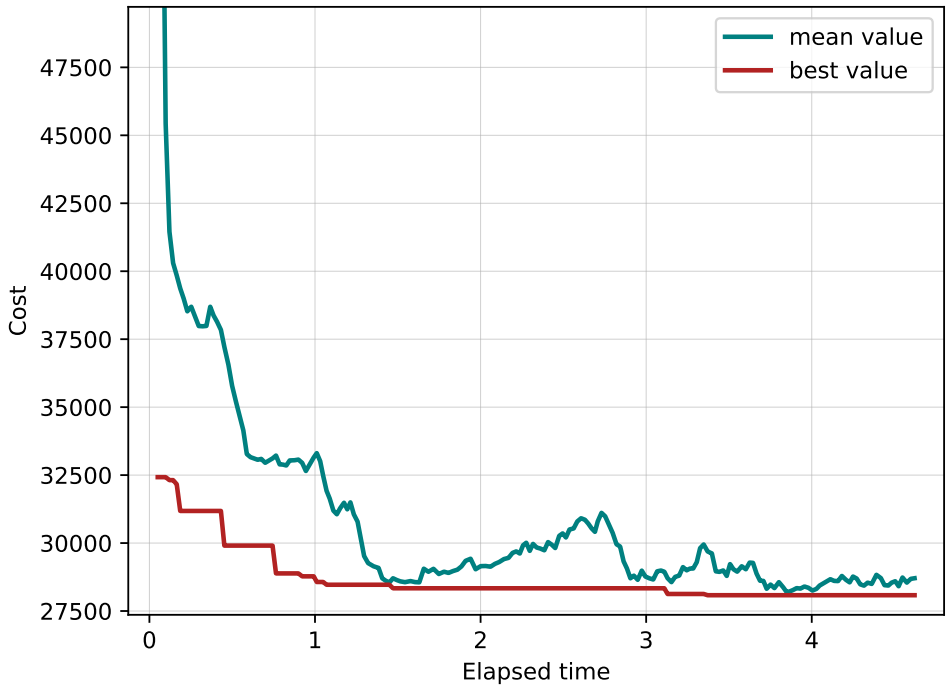
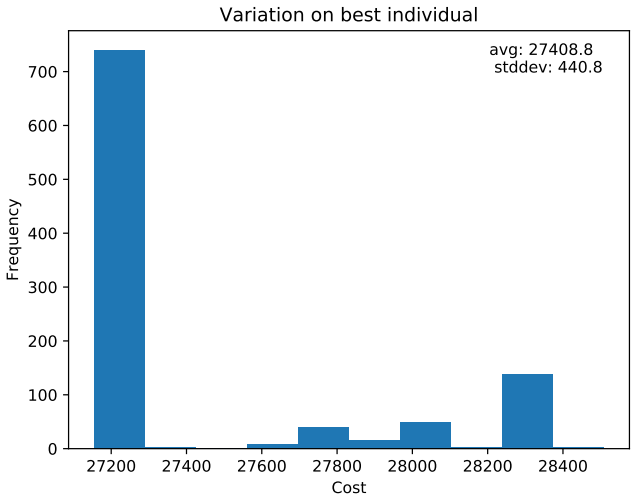
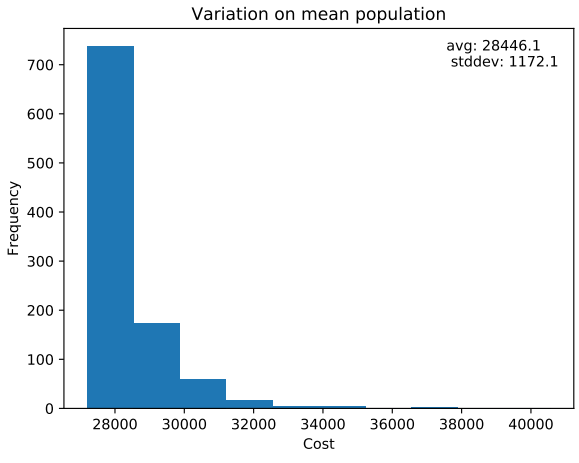


Figure 5: Typical convergence graph of tour29.csv

Figure 6 shows the histograms that are obtained when running the problem 1000 times with the GA. It can be seen that the algorithm finds the optimal solution in 3/4 cases. Sometimes it stops with a not-optimal value that is close to the optimal one. From Figure 6b, it can be seen that the cost of the mean population usually is quite higher. There may be a little more exploitation needed at the end of the algorithm in order to lower the average cost of the population and to find the optimal solution every time. Now, the algorithm often gets stuck in a local optimum of ± 28300. Nevertheless, without a stop criterion the algorithm always finds the optimal value within 5 minutes.



(a) variation on the best individual



(b) variation on the mean individual

Figure 6: Histograms that shows the variation on the best individual and mean individual when the problem is run 1000 times

4.3 tour100.csv

A typical convergence graph of this problem is shown in Figure 7. It can be seen that the algorithm converges too quickly after 50 s. At this point, the algorithm does not find a better solution and start searching in a wrong direction (hence the increase in mean cost). The spikes on the mean cost are due to tours with disconnected cities. The cost of these tours has been set very high, which causes the spikes in the mean population.

The lowest cost that has been found with this algorithm is 7521. This is better than the heuristic value of 8637, but still 200 higher than the optimal value of around 7350. The hyperparameters are probably not optimal for this problem.

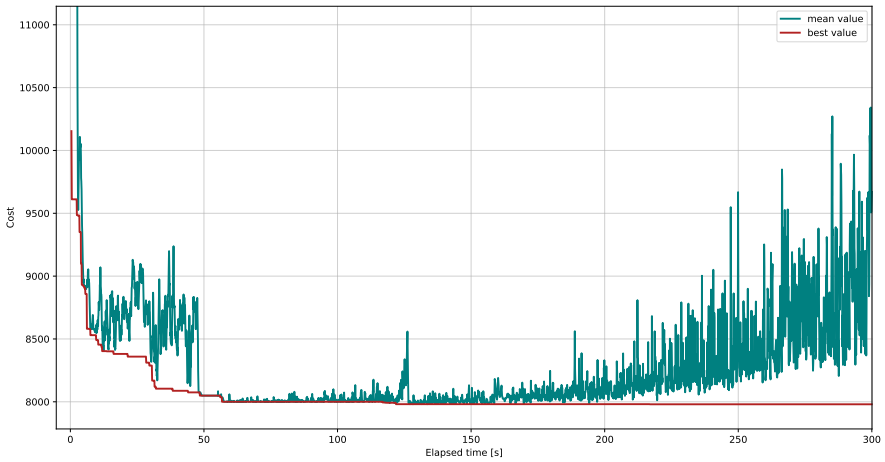


Figure 7: Convergence graph tour100

4.4 tour194.csv

Figure 8 shows a typical convergence graph. The best-found tour length is 9843. The diversity here is good, as the mean cost always is a bit higher than the best cost. The algorithm finds in 5 minutes a value that is clearly better than the heuristic value (11385). The optimal value is 800 less, so the algorithm still can be improved to find a better solution in five minutes.

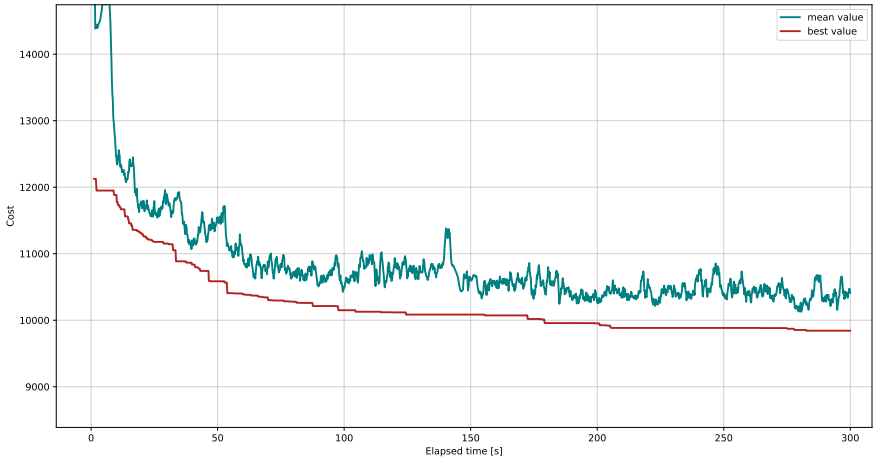


Figure 8: Convergence graph tour194

4.5 tour929.csv

Figure 8 shows a typical convergence graph. The best-found tour has a cost of 112205. This is slightly better than the heuristic value (113683). However, the optimal value of 95300 is still far away. The algorithm does

not perform well on bigger tours because the parameters are not optimized. The diversity is too high and the algorithm does not converge to an optimal value. The algorithm is too explorative here. Also, the algorithm is a lot slower on these bigger tours and less iterations can be done in five minutes.

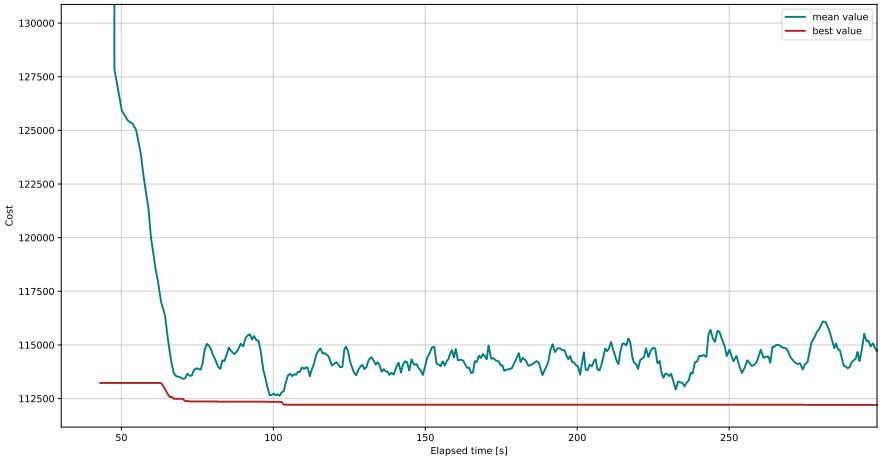


Figure 9: Convergence graph tour929

.

5 Critical reflection

The main strengths of a GA are in my opinion that it can be used in various representations and thus problems. It can come up relatively fast with good solutions and the basic concept is easy to understand. In my algorithm, I also tried to optimize the code such it can run iterations fast and come up with good solutions for this problem in a short period. I think the main weakness of a GA is the tuning of the different parameters. This is also the main weakness in my algorithm. There are a lot of parameters to be considered (k in k-tournament, α for mutation, λ for population size, μ for amount of offspring, ...). The parameters are chosen based on tour194 but may be not the best for the others. It is time-consuming and sometimes difficult to find out what the best values for these parameters are. One can try to implement self-adaptivity, where the different parameters adapt with every new generation. But there is no 'one size fits all' solution and self-adaptivity is still subject of many studies today.

An evolutionary algorithm is very appropriate for the TSP in my opinion. A brute force approach would be way too slow because of the very high number of possible tours. The algorithm can also build on existing heuristics for this problem to come up with even better solutions.

I thought it was cool that this course is very hands-on. It did certainly improve my python programming skills. Before I did not know about the existence of GA's and now, I am able to set up a (basic) genetic algorithm. The drawback is that the study load of this course was quite high during the semester.

References

[1] O. Abdoun, J. Abouchabaka, and C. Tajani, "Analyzing the performance of mutation operators to solve the travelling salesman problem," 2012.

[2] A. E. Eiben and J. E. Smith, *Introduction to Evolutionary Computing*. Natural Computing Series, Berlin, Heidelberg: Springer Berlin / Heidelberg, 2015.

[3] B. Freisleben and P. Merz, "A genetic local search algorithm for solving symmetric and asymmetric traveling salesman problem," pp. 616 – 621, 06 1996.