Evolutionary Algorithms: Final report

Castellucci Davide 0834601

January 4, 2021

1 Metadata

- **Group members during group phase:** Ioannis Mouratidis and Cedric Mousin
- **Time spent on group phase:** 10 hours
- **Time spent on final code:** 50 hours
- **Time spent on final report:** 10 hours

2 Modifications since the group phase

2.1 Main improvements

In the individual phase I tried several new functions in order to improve the previous project made in collaboration with my group, some of them weren't effective, others had some pros and some cons and others definitely led to better results. The list of the attempts will be explained in the dedicate subsection, here i will present only the ones that turned out to be successful:

- General optimization of the code
- Insertion of new data structure 3.1
- Insertion of a new mutation operator 3.4
- Modification of recombination operator 3.5
- Modification of elimination operator 3.6

Short description 1: The general optimization of the code has been done done in all the code by optimizing the existing function removing or reducing redundant cycles and also by optimizing how and when to calculate the scores of the paths, avoiding to calculate them multiple times, since for big datasets this require quite a lot of time. This has been done by calculating the score of the initial population and then the score of a children or a mutated individual as soon it was calculated and saving them in a datastructure.
The second improvement was the introduction of this datastructure: a dictionary in which the keys were the scores and the values were the relative path, this allowed a faster in finding the best solution and avoids to calculate 2 times the score for the same path.
The third improvement was the implementation of the sequential constructive crossover (SCX). This was by far the best improvement since it allowed to speed up the process and obtain very good results for all the datasets.
The fourth improvement was the insertion of a new type of mutation operator in order to improve the probability of escaping from local optima. The basic idea of this mutation operator is to swap two subpaths instead of just two cities.
The fifth improvement was to modify the elimination operator eliminating identical individual in the subsequent population in order to maintain diversity in the population.

2.2 Issues resolved

The main issues that we encountered in the group phase were:

- Slowness of the code: especially for big datasets the code didn't run in an acceptable amount of time and didn't reach good results. This issue as already told was solved by optimizing the code.
- Local optima: the algorithm also with small dataset got stuck in local optima (with score around 29000 for tour29). This was solved by introducing the new mutation and recombination operators allowing the algorithm to reach way better results than before. Almost every time the algorithm reach the global optima for tour29 and if not it's still very close to it (with scores never never bigger than 29000).

- Lack of diversity: after some cycle in the previous code the population started to be composed by the same individuals and for this reason also the children generated were always the same and this didn't allowed an appropriate exploration of the state space. I solved it by modifying the elimination function as described above.
- Adaptation of algorithm for missing links between cities: this issue has been solved by reading the matrix and substituting the 'inf' cells with 2 times the maximum distance and then letting the algorithm to deal with it (to note that for how is structured the algorithm it will never chose to link two of these cities).

Short description 1: All the issues listed in the group phase were solved in the individual phase and moreover several other attempts were made introducing other functions that also reached the goal of solving these issues but the best overall solution was the one presented so far. This allowed to drop substantially the cost of the path especially for big dataset.

3 Final design of the evolutionary algorithm

In the algorithm that i presented i choose to use a random initialization of the population of 100 individuals, followed by selection of the parents with k-tournament selection and the generation of 100 children thorough SCX, then two types of mutation (swap mutation and a variants of it) and in the end the $\lambda + \mu$ elimination, while as stopping criterion the algorithm require to obtain the same result 100 times in a row or to run out of time

3.1 Representation

In the group phase we decide to represented the population as a numpy array composed by lists of path. This representation worked already well due to the built-in function in numpy, but i also decide to introduce a dictionary to keep the pair score:path in order not to have to calculate again the length of a path if it was already in the dictionary and to speed up the process. In order not to make this dictionary become too big and occupy a lot of memory, when a new population is generated only the pair score:path of the new population are kept. This also allowed to search between the paths basing on their scores and also that thing allowed to speed up the search, since instead of comparing N lists (with a number of elements up to 929) the algorithm has just to compare N floating point numbers. Another consideration was to use integers instead of floats by transforming all the elements in the matrix in int in order to exploit the speed-up in using integers during the computations instead of float introducing a very small approximation, but this idea has been discarded since in the end the gain in terms of time was little or even null.

3.2 Initialization

I spent a lot of time on the initialization by trying several initialization methods with varying levels of heuristics in them, but in the end the one that gave better results was the random initialization in the population since it guarantee to maintain diversity in the initial population allowing the algorithm to converge faster and not being stuck in the local optima. I made this decision since i noticed that after more or less 20 cycles the algorithm with random initialization usually started to give better results than the algorithm with heuristic initialization; this was a problem only with the biggest dataset tour929 since in 5 minutes it didn't have the time of computing 20 cycles but anyway the results were comparable (around 11500 instead of 11200) but also using a random initialization it was more appreciable the work of the genetic algorithm since in the first steps the drop of the loss function (length of the path) were very big while with the heuristic initialization were very small.
In the code are proposed 3 heuristic/probabilistic initialisations:

- The first one is a pure heuristic initialization: a random city is selected and then from that one the path is generated by choosing the next city to visit as the closest to that one that hasn't been visited yet. The diversity in the initial population was guaranteed from the fact that the first city was different so the list of non-visited cities was different every time. This initialization was the one that led to better results between the non-random ones and the results were almost the same as with the random initialization, it performed slightly better in the tour929 and slightly worse in the smaller problems.
  The strong point of this initialization was to start with an already good solution and don't have to wait several cycles to arrive to it, but the drawback is to pay something in terms of diversity of the population.
- The second attempt was made in order to try to keep more diversity and start in a good position. The initialization was a combination of heuristic and probabilistic initialization in which at each step a probability of being selected was assigned to each non-visited city basing on their distance from the actual one.
  The result with this initialization were very good for small datasets, but it became too time consuming in the tour929 in which it consumed more than half of the available time since it has to compute every time the probability of each city of being selected.
- The third attempt was a variant of the second one in order to exploit the built in function in numpy library "numpy.percentile". The idea was to divide the non-visited city in 10 sets using the 10 percentile on a row

of the distance matrix. In the first set there were only the cities in the first 10 percentile, then in the second one there were the one in the first plus the one in the second 10 percentile and so on so far, up to the 10th set in which there were all the non visited cities. Then a random set was chose and in this set a random city was selected. Doing so the cities in the first set (namely the 10% of the cities that were closer to the actual one) had 10 times the possibility of being chosen with respect to the most distant ones. Unluckily this initialization led to worst results than the other 3 presented and it has the only advantage of working fast enough also for the tour929.

3.3 Selection operators

The selection method that I chose was the k-tournament selection, that was the one chosen during the group phase. The main changes was to remove an inner loop in which the length of the paths were calculated because, since I introduced the dictionary in which the length were already stored, there were no need of computing it again. This allowed a big speed-up in the big datasets since this cycle was computed one time for every generated children at each iteration. The parameter that i used in this function was the number of possible candidates and i chose 5 possible candidates between which chose the parents, i chose this parameter because during the group phase we already did a study for the parameter setting on this function and this was the parameter that was performing better and repeating these experiments with the new implementation i obtained the same results.
I also considered and implemented roulette wheel selection but it performed worse and it was more time consuming.

3.4 Mutation operators

I decided to use 2 mutation operator in my project. The first one is the swap operator that was implemented in the group phase, then I decided to introduce also a variant of it since i considered that for big datasets it was unlikely to swap exactly the two cities that improved the solution, so I changed it by swapping two sub-permutation in the path, in which starting point and the length of the two sub-permutation were random (obviously constrained in order not to overlap and go outside of the array).
A further improvement that i thought for this mutation operator is to chose as breaking points for the sub-permutation the link between cities with bigger distance, but I didn't implement this because in the end it already performed well and I preferred to spend the time at my disposal to implement better recombination operators since it was the change that most improved the solution.

3.5 Recombination operators

As recombination operator I chose to use the sequential constructive crossover because, looking in the literature, it seems to be the one that perform better and my results confirmed that because between all the recombination operator that I implemented, it was by far the one that went closer to the optimal solution. The idea behind this operator is to construct an offspring using better edges on the basis of their values present in the parents' structure. It also uses the better edges, which are present neither in the parents' structure. The SCX does not depend only on the parents' structure; it sometimes introduces new, but good, edges to the offspring, which are not even present in the present population. Here i will present the algorithm.

1. Start from a random node of one of the 2 parents.
2. Sequentially search both of the parent chromosomes and consider the first 'legitimate node' (the node that is not yet visited) appeared after 'node p' in each parent. If there is no 'legitimate node' chose the node with lower distance from the actual one.
3. Suppose the 'node α' and the 'node β' are found in 1st and 2nd parent respectively
4. If $c_{p\alpha} < c_{p\beta}$, then select 'node α', otherwise, 'node β' as the next node and concatenate it to the partially constructed offspring chromosome. If the offspring is a complete chromosome, then stop, otherwise, rename the present node as 'node p' and go to Step 2.

Another recombination operator that I implemented was a variant of this in which the basic concept was the same, but also depends on a critical distance calculated offline for each city with the formula:

$$d_c{}^i = \frac{1}{B(N-1)} \sum_{j=1}^{N} d_{ij}, B \geq 1$$

. Then the algorithm is the same except for the 3rd step in which the closest of the four 'legitimate node' (if any) is compared with the critical distance an if the distance between the 2 node is bigger than the critical distance, the node with lower distance from the actual one is selected. This algorithm worked more or less as good as SCX but it was a bit slower so it was discarded. Finally I implemented also edge recombination, that actually was the first

attempt for the individual phase after the implementation of order crossover in the group phase, but these gave very bad results compared with the ones obtained with the others 2. The operator that I selected is good because exploit the good features of the parents since it always chose the best one between the neighbour nodes of them, allowing changes in the case of the parents overlaps a bit, since the mechanism of the 'legitimate nodes' allow to chose another node that is not a neighbour in none of the two parents.

3.6 Elimination operators

As elimination operator I maintained the $\lambda + \mu$ elimination and i changed the one implemented in the group phase by not allowing the possibility to retain identical individuals in the next population, this was useful in order to maintain diversity and not converge to local optima. I chose to maintain this one because it seemed to work already quite well, while using some type of probabilistic elimination would have led to a waste of time and especially in the tour929 it would be a problem since it already compute very few cycles and if it was slower it wouldn't have computed enough cycles to arrive to acceptable solutions.

3.7 Local search operators

I didn't implemented a proper local search operator, the only part in which a bit of heuristic was used was when in the SCX there was not a 'legitimate node' and the algorithm searched for the closest one. I made this decision since, from the initialization i saw that the use of an heuristic lead to a lack of diversity and i wanted to maintain it in order to make the genetic algorithm work well. Moreover whatever type of heuristic imply that the program spend time and resources computing it so i preferred to spend all the resources to compute more cycles in the genetic algorithm

3.8 Diversity promotion mechanisms

The only diversity promotion mechanism that i used, as already said was the one in the elimination function that doesn't allow to retain identical individual in the next population. I think this was enough to have a sufficiently high diversity in the population since it is compose by 100 different individuals

3.9 Stopping criterion

As a stopping criterion i chose to stop when the same result was obtained 100 times in a row since this gave time to the algorithm to escape from local optima by finding other better solution. The other stopping criterion was, of course, the one implemented in the Reporter class in which the limit is the time (300s).

3.10 The main loop

In the main loop I followed this sequence:

1. Initialization of the population
2. Evaluation of the initial population
3. While not termination condition:
4. For n in number of generated children:
5. Select parents
6. Generate child
7. Evaluate child
8. First mutation
9. Second mutation
10. Elimination
11. Evaluation of stopping criterion
12. Registration of the best solution

I decided to apply operators in this order because i wanted to apply the mutation operators on the whole population, including the generated children

3.11 Parameter selection

The choice of the hyperparameters were made empirically by launching the program changing one of them at the time on the different datasets. The best trade-off between speed of the algorithm and performance was to have a initial population of 100 individuals and a generation of 100 children and a probability of mutation of 10% for both of the mutations. The full study with all the results varying the parameters can be found in the report of the group phase and the experiments made in the individual phase confirmed that results. This hyperparameters

penalize the big datasets since it take a bit of time to compute one cycle on a population of 100 individual and they work slightly better with a population of 50 individual and 50 generated children, but this would decrease the diversity and in the smaller datasets there would be a risk of being stuck in local optima, but in general the choice of this hyperparamenters didn't affect excessively the solution, except in the case of pushing these to the limit and have very small o big population size

3.12 Other considerations

The last consideration was an attempt that i didn't have time to implement successfully. The idea was to try to speed up the computation in the bigger datasets by grouping near cities with a sort of clustering, optimize the path between them and then find the closest cities in 2 different cluster and chose them as breaking point to link the clusters. This because if there are a lot of cities very close between them, it's very likely that they are connected together in a sub-path, but i abandoned the idea because it was effective only for datasets in which some clusters can be clearly identified. Up to that moment i was able to implement a different type of clustering that wasn't neither DBSCAN or Kmeans, because the first one is not really effective with datasets with a lot of variability in the distances between points (that was the case with these datasets), while the second one require as input the number of clusters or it need an iterative process to find the optimal number of clusters. So the clustering method that i used was based on the principle of neighbourhood of the DBSCAN, but the radius ϵ of the sphere was function of the critical distance of that point. With this algorithm I were able to find some clusters in some cases (like in tour29) but in most of the cases i found 0 up to 2 little clusters and this wasn't really effective in speeding up the optimization process and in most of the cases it led to worse performance.

4 Numerical experiments

4.1 Metadata

The hyperparameters in the algorithm were the ones listed before. For each dataset, as mentioned before the optimal set of parameters were a bit different but I didn't implemented the variability of the parameters in function of the information from the problem because I think it would have been too specific for the 4 datasets that we have and if launched on another benchmark problem it could lead to worse performance, so I just chose the best trade-off between all parameters . The program was run on a computer octacore with processors 'Intel core i7-7700HQ', 'CPU 2.80GHz' and 16GB of RAM, in a PyCharm environment that uses as interpreter Python 3.7.

4.2 tour29.csv

The implemented algorithm work really well for the torur29 dataset as can be seen in the plot, it converge really fast and in most of the time it converges to the optimal solution, that is 27154.

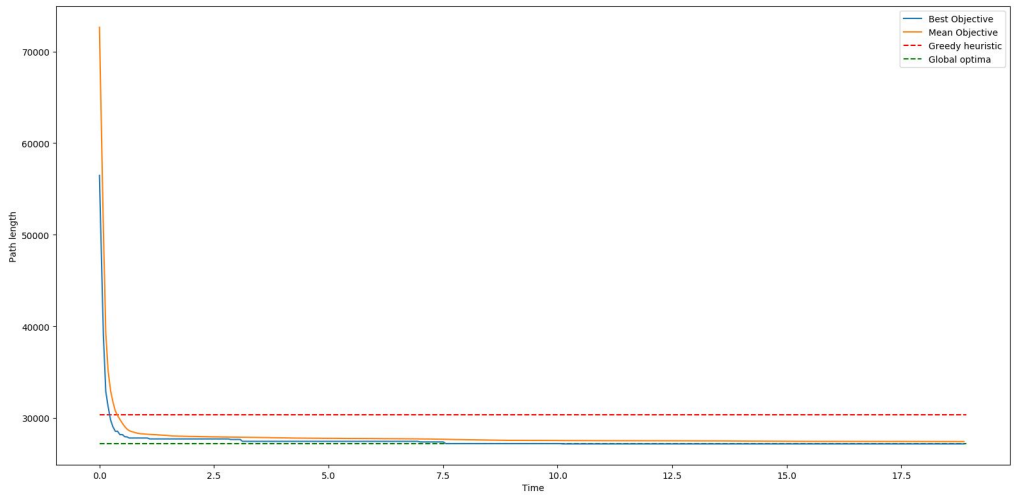

Figure 1: Length of the path in function of time.

In the figure 1 it is possible to observe how the algorithm goes under the level of a simple greedy heuristic after some milliseconds and spend the rest of the time approaching the optimal solution, but maintaining for almost all the time values very close to it.
The best tour length that i found is the global optima and it correspond to the tour [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 5 0 1] or a whatever shift of this one. To have a statistically significant result I

launched the algorithm 1000 times obtaining the results that are shown in figure 2, 3, 4 and in the table.

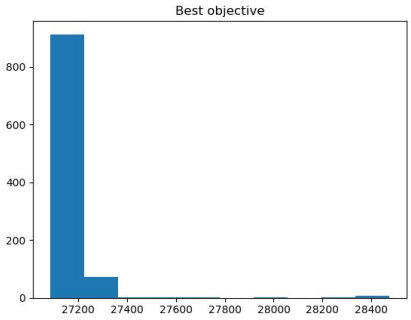

Figure 2: Hist of the best objective

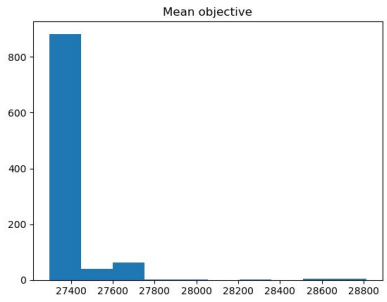

Figure 3: Hist of the mean objective

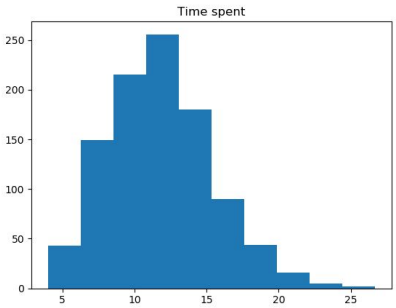

Figure 4: Hist of the time spent

	μ	σ	min	max
Best objective	27200.9	150.6	27154.5	28543.9
Mean objective	27450.1	200.2	27381.6	28829.3
Time	12.9	3.5	4.8	26.3

By looking at these result it's appreciable how more than 80% of the times the algorithm find exactly the optimal solution and it only goes above 28000 very few times. This explain the reason of a pretty high variance since for the 3σ rules the 99.9% of the results should be in the interval $\pm 3\sigma$. The same conclusion can be done for the mean objective function, this means that all the population (that is forced to have different scores) approach the optimal solution. Regarding the time it its quite high for solve such a simple problem but this is because of the stopping criterion, indeed it if would be lowered to require the same result 50 or 20 times, the algorithm would finish in less than half of the time, but this would compromise the performance allowing it to reach the global optima less times.

4.3 tour100.csv

For the tour100 the algorithm still work quite well, indeed as it is shown in the figure 5, it can be noticed how after some seconds it gets better result than a simple greedy algorithm and then it approaches the global optima, but without reaching it in this case, even if the solution is quite close to it.

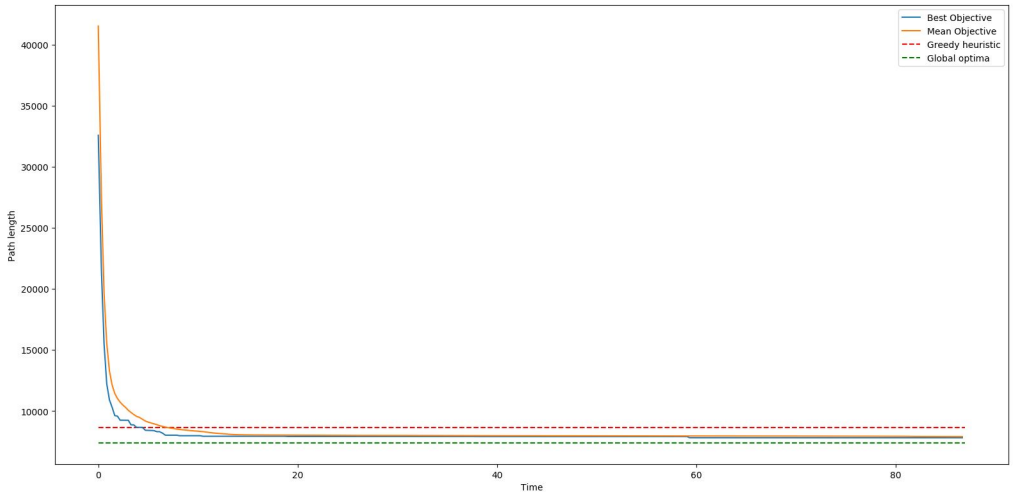

Figure 5: Length of the path in function of time.

For this and bigger datasets I didn't have time to launch it 1000 times so I did it just 10 time, in order to have some statistical data and plots, even if not really relevant from a statistical point of view. The obtained results are shown in the figure 6, 7, 8 and in the table as before.

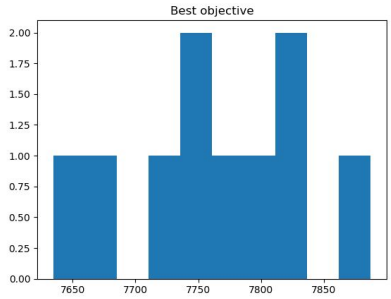

Figure 6: Hist of the best objective

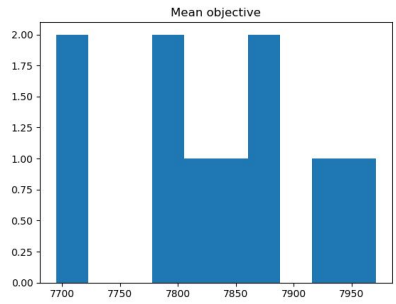

Figure 7: Hist of the mean objective

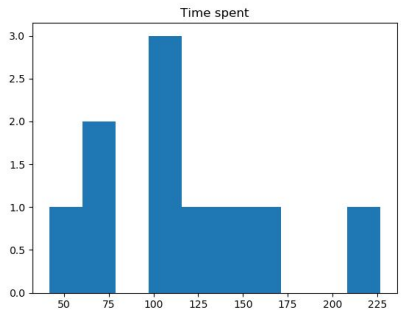

Figure 8: Hist of the time spent

	μ	σ	min	max
Best objective	7772.9	70.1	7647.1	7899.8
Mean objective	7839.3	83.5	7708.4	8064.1
Time	124.2	49.4	50.8	236.2

Also in this case best and mean objective are close because the stopping criterion give the time to all the population to approach the best solution. Regarding the time it can be noticed in this case how, increasing the dimension of the dataset, it also increases more or less linearly. This result can be quite surprising since this is a well known NP-complete problem and, because of this, the time should increase exponentially, but it can be explained by the fact that the solution found is not the optimal one and so the problem is not really solved.

4.4 tour194.csv

Also for tour194 the performance are not bad, it always beat the greedy heuristic (even if should be noticed that the greedy heuristic requires way less time) and approach the optimal solution being on average 10% bigger than it. The result are shown in the figure 9

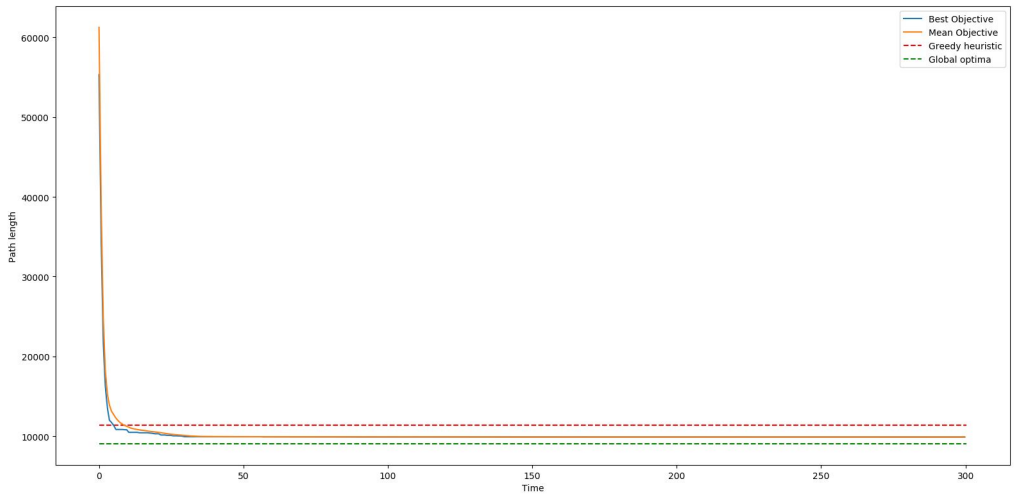

Figure 9: Length of the path in function of time.

The same considerations made before can be done also in this case launching the algorithm 10 times. The results are shown in figure 10, 11, 12 and in the table

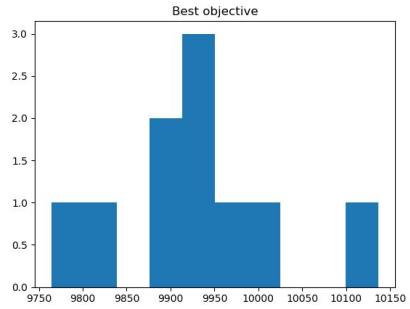

Figure 10: Hist of the best objective

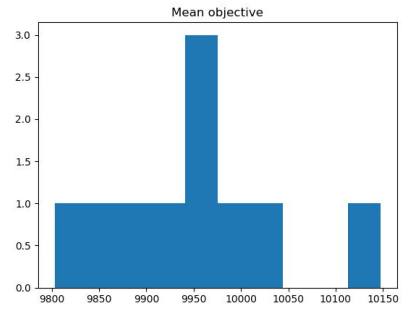

Figure 11: Hist of the mean objective

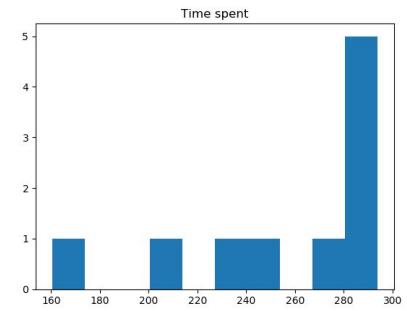

Figure 12: Hist of the time spent

	μ	σ	min	max
Best objective	9947.0	95.2	9782.6	10155.7
Mean objective	9963.4	91.8	9820.2	10165.0
Time	265.0	44.4	167.0	300.0

Starting from this dataset the maximum time begin to be 300s that is the amount of available time, this means that the algorithm would have needed more time to find the best solution according with the stopping criterion.

4.5 tour929.csv

For this benchmark problem the algorithm starts to function worse than a greedy heuristic and it requires a lot of time to compute each cycle as shown in figure 13. Indeed it compute approximately 15 to 20 cycles and it doesn't have enough time to improve the solution, this because this happens after 20 cycles for my algorithm. Anyway the solutions can be considered acceptable since they are 15% - 20% bigger of the optimal solution with a very big dataset.

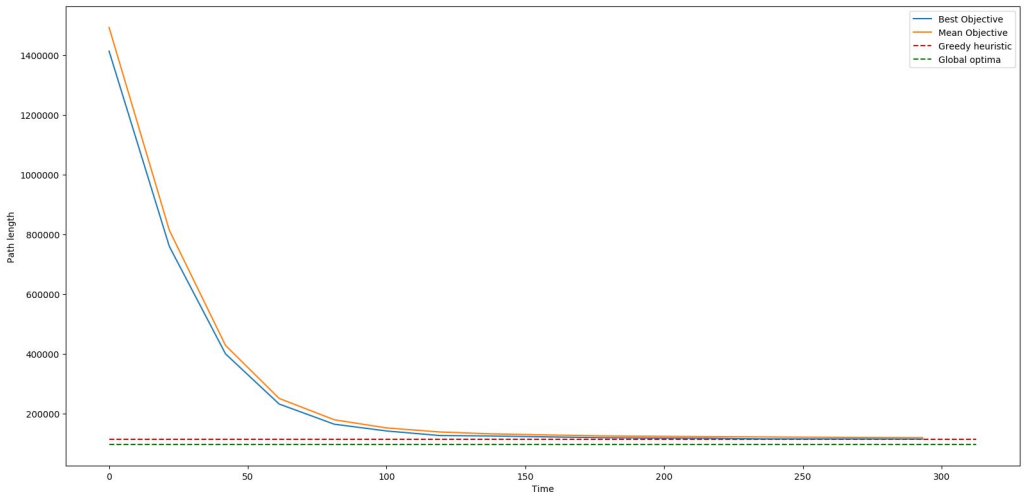

Figure 13: Length of the path in function of time.

By launching the problem with an heuristic initialization, it already started close to the level of the greedy heuristic but it can not be appreciated so much the decreasing behaviour of the loss function, this can be seen in the figure 14.

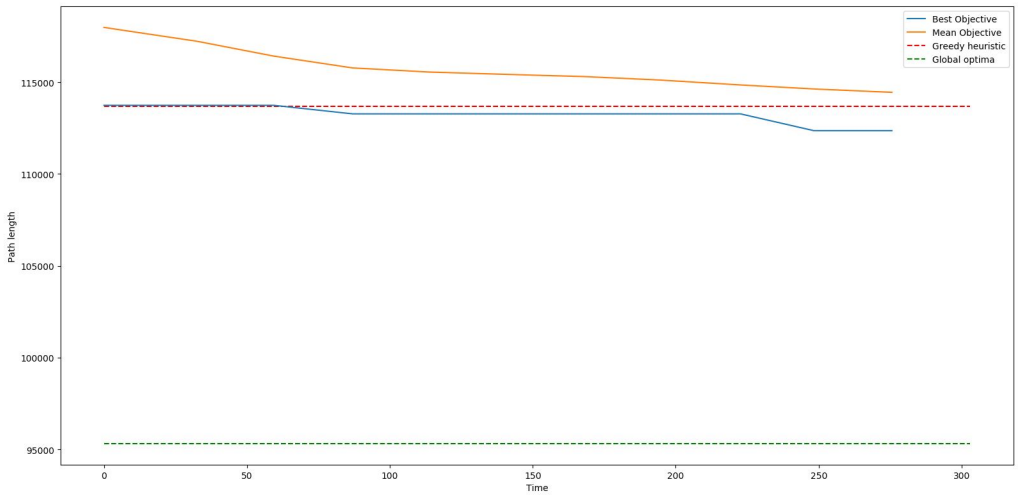

Figure 14: Length of the path in function of time.

From the histograms in figure 15, 16, 17 and the table, it's possible to observe how in this case the mean objective it's quite higher than the best one, this means that the population was still adapting to the best solution when the time expired. This confirm the hypothesis that the algorithm would need more time to find a good solution and get closer to the optimal one. Regarding the time the distribution has absolutely no meaning because all the runs make use of the available time and the different values are because the 'break' in the 'while' loop is at the end, so it depends on when the last cycle starts.

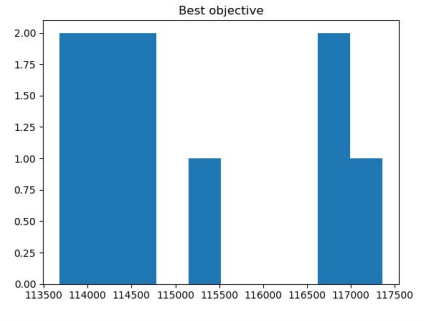

Figure 15: Hist of the best objective

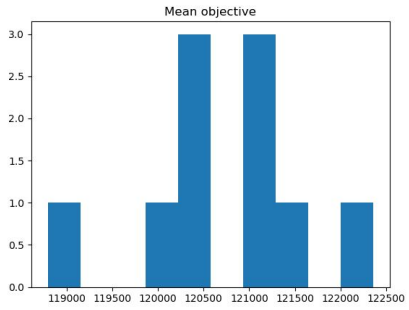

Figure 16: Hist of the mean objective

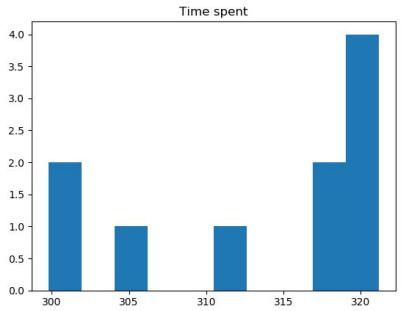

Figure 17: Hist of the time spent

	μ	σ	min	max
Best objective	115356.4	1280.3	113860.2	117546.1
Mean objective	120943.8	889.9	118970.8	122539.6
Time	314.5	8.3	300.8	322.2

5 Critical reflection

The strong point of using the evolutionary algorithms is the capacity of combine exploration of the search space and exploitation of the features of good individual. In my project I tried to use at best this 2 features, for the exploration implementing 2 different mutation operators and for the exploitation using k-tournament selection and SCX, in particular the crossover technique allow to exploit the best components of both the parents selected and it's not just a blind recombination operator like others that I used at the beginning in this project. The main week point that I encountered for my code is the time required for finding good solutions and the difficulty of finding the global optima for big datasets, as an example for the tour929 the algorithm wasn't able to reach the results that a very simple greedy heuristic find in less than 1sec. The obvious mitigation of this problem would be to start in an already good position initializing the problem with an heuristic, but this would raise the problem of the loss of diversity that is fundamental for the correct functioning of the genetic algorithm.
I think that the genetic algorithm is well suited for this problem because of it easy representability as an individual

of a genetic algorithm and also the structure of the problem allow to implement easily crossover and mutation operators allowing efficient computation.
Personally the most surprising thing in this project is to see in practice that a random or semi-random process actually converge toward the optimal solution and it's incredible to think how this happen spontaneously also in the nature.
From this project i learned, beside the concepts of the evolutionary algorithms, that to combine different strategies can lead to better result than just blindly applying the one that seems to be easier or to function better.

6 Other comments

The last comment that I would like to do for this project is that i noticed that the distance matrix is not symmetric, namely the distance from city A to city B is not the same as the distance from city B to city A. This could be caused by uncertainty in the measurements and imply that run across the path in one direction or in the other lead to different length.