Evolutionary Algorithms: Final report

April 23, 2025

1 Changes since the group phase (target: 0.25 pages)

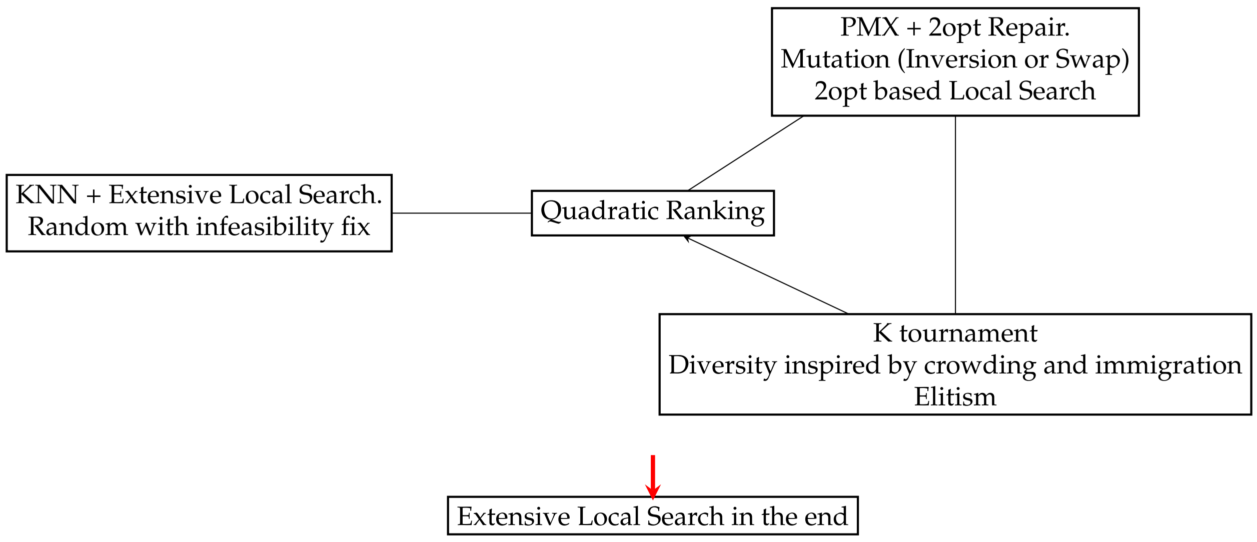
1. Advanced initialisation where a part of the candidates is picked through heuristics. The rest (majority) is randomly initialised and fixed so that they are not infeasible (see 3.4).
2. Fixing candidates that are created during variation in case they are infeasible (see 3.9 - two opt repair).
3. Local search based on two-opt operation. Diversity promotion inspired by immigrants and crowding (3.10).
4. Numba optimised code, much better performance.
5. Mass Mutation in the end (extensive local search), to better approximate local minimum (3.13).

2 Final design of the evolutionary algorithm (target: 3.5 pages)

2.1 The three main features

1. Advanced Initialisation.
2. Extensive Local Search to better approximate local minimum.
3. Two opt based repair mechanism.

2.2 The main loop


PMX + 2opt Repair.
Mutation (Inversion or Swap)
2opt based Local Search
KNN + Extensive Local Search.
Random with infeasibility fix
Quadratic Ranking
K tournament
Diversity inspired by crowding and immigration
Elitism
Extensive Local Search in the end

A lot of information regarding the choice of design is also discussed on a later section, where I outline the weak points of my algorithm and suggest improvements.

2.3 Representation

Answer same as intermediate phase: To represent the candidate solutions we decided to use a representation based on permutations. This seemed like an intuitive choice for modelling the cycles and we knew from the lectures that using permutations is very frequent in mathematical optimisation and evolutionary algorithms - therefore we could easily study about the variation techniques used in this case (which is very important in the design of the algorithm).

To be specific, each candidate corresponds to one 1D numpy array, where the ith element (an integer) corresponds to the index of the city to be visited in the ith step. The city index comes from the csv files. The array is of size num_cities +1, where the first num_cities elements are the tour and the last one is the same as the first one to convey the property of the cycle. The population is a 2D numpy array where every row is one candidate.

2.4 Initialization

The vast majority of individuals to be initialised are initialised randomly. This is done because the algorithms needs diversity, especially in the beginning, to be able to function properly. To get the random candidate, I just find a random permutation and make sure that the last element is equal to the first.

In larger datasets it is very important to fix the infeasible candidates that come from the process I just described. To fix them I essentially use the two opt repair operation (described on 3.9).

Apart from diversity it really helps if we help the algorithm by giving it some good initial candidates. To achieve this, I essentially run a DFS algorithm with a greedy logic, which iteratively constructs a candidate one city at a time.It is essentially DFS with backtracking, but restricted to the k nearest neighbors at each step. At every node, it takes the k-nearest unvisited cities (the nearest neighbors). It tries the first neighbor, pushing both a backup state with the remaining neighbors for later backtracking, and a new state continuing the path with that neighbor to a stack. It then continues down that new path until it either completes a feasible tour or runs out of moves. Backtracking is important to be able to find feasible solutions without spending too much time.

Apart from these, I found that larger datasets have an even larger need of good candidates in the beginning. To achieve even better initial candidates, I pick the best element that I get from the knn initialisation and constantly apply mutation for 10 seconds to get a good choice (more on the process of mass mutation on 3.13). I do it with both swap and inversion mutation - this way I also estimate which one of the two methods to use later by picking the one that gave me a better final candidate. I do not add the mutated candidates in smaller datasets because there is no need to and I found that it messes up with the earlier diversity.

Smaller datasets have 50 knn initialised candidates while larger ones have 250. These are values that I found gave decent results for the experiments, They are under no circumstance optimal nor where they calculated by any sophisticated technique. To make sure that the population enrichment scheme does not take over the population I also use a diversity promotion technique. I use a dataset size of 1600 candidates throughout the experiments. This size was basically picked with the performance of the larger datasets as a criterion - the smaller ones could easily be solved with smaller populations.

2.5 Selection operators

Answer same as intermediate phase: The choice for selection and elimination happened simultaneously. We decided not to use the same technique for the two separate steps, because it might become too biased towards good performing individuals (higher probability to pass through both of the selective steps), which might reduce diversity. In the class we discussed about two major categories of such algorithms, fitness based and competition based methods.

Elimination seems like a stricter process, as it can completely erase the information of one candidate for the next generations. On the other hand, selection should be less strict to give emphasis on the fact that we should allow candidates that are bad (in terms of obj value) to give offspring and increase diversity. As the generations go on, the strictness can increase, but in the early stages exploration is crucial.

For selection, we decided to go with ranking based methods. These methods have the advantage that it is fairly easy to control the strictness (through $s = \alpha^{gen\_num}$). We can start by choosing $a \simeq 0.995$. Values near 1 are reasonable. We chose the quadratic method because the linear might have been too lenient on the later stages.

For elimination, we used an algorithm from the other category, specifically k-tournament. We chose this one because it is simple to implement and it generally gives decent results (e.g. as seen on the hands-on programming sessions). A smaller k (e.g. 3) gives a smaller selection pressure than a larger one (e.g. 10). The reasonable values are generally inbetween and near these values.

2.6 Mutation operators

In the beginning I just used the inversion mutation. I also wanted to try another mutation technique and also incorporated swap mutation in the algorithm. To estimate which one to choose I use a trick that is described on the initialisation part. It turns out that some datasets (especially larger ones) prefer swap mutation, while others prefer inversion. I found empirically that swap mutation might introduce more randomness. It is also worth noting that after mutation I also apply a local search mechanism based on two opt. The two opt operation is similar conceptually to inversion which might make the need for a different mutation operator more prominent.

2.7 Recombination operators

Answer same as intermediate phase: Similar to the mutation operator, we studied the book to choose this operator. We saw that for problems like the TSP, the PMX and the Edge crossover operators are two classic choices. We do not describe the way they work due to constraints on the size of the report, we let the book cover us on this. On a first glance PMX seems simpler to understand and implement.

I continued by using the PMX operator. I fix infeasible offsprings by using the two opt repair technique described below. More comments on the choice of recombination operator can be found on section 6.

2.8 Elimination operators

Answer same as intermediate phase: For the elimination step most of the information has been given on the section about selection. We should highlight the fact that we have included elitism (i.e. the most fit individual always survives) to make sure that we do not lose the best solutions. In this phase I also include a diversity promotion mechanism exactly after the elimination .

2.9 Local search operators

To implement local search, I basically wrote a function that uses the 2 opt swap (takes a candidate and two edges, reverses the part in between the edges) to fix infeasible candidates. This method picks a disconnected edge and searches onward on the candidate solution for edges that if they were to be swapped the infeasibility would be removed. The process is done again on the changed candidate. If there are no infeasible edges, the process is succesful. In other cases (there is a maximum amount of attempts allowed) the algorithm returns none.

This process is used both to fix infeasibility but also to apply local search. After mutation, I apply the two opt repair process >1 times and find the fixed candidate with the best objective value - this is the one I accept for the next step. This way I explore locally to improve the candidate a little bit.

The extensive local search I do at the end (but also on the best knn initialised candidates) is slightly different. I basically apply mutation to the candidate a lot of times and change the candidate every time I find a better value (mass mutation).

2.10 Diversity promotion mechanisms

In this part of the algorithm, I wanted to design a diversity promotion mechanism that would be able to offer enough diversity, without demanding excessive change in the overall algorithm design. The fact that I do not implement any sophisticated hyperparameter choice step also made me prefer a mechanism that is simple in terms of hyperparameter choice.

I decided to design a mechanism based on immigration. To be specific, before ending the elimination step, I replace a specific amount of the candidates (e.g. 1%) with new random ones (immigrants). I put emphasis on getting rid of duplicate candidates first - this was done after inspiration from the idea of crowding where we get rid of similar elements (in our case identical). If the amount of immigrants to enter the population exceeds the amount of duplicates, I replace unique candidates but I forbid the mechanism to get rid of the 10 percent best ones, in an attempt to better preserve promising candidates.

2.11 Stopping criterion

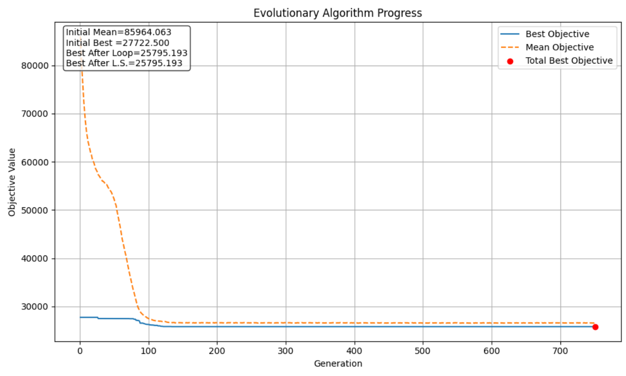
I essentially do not use a stopping criterion, because I wanted the algorithm to take advantage of the whole 5 minutes that are available. It is worth noting however that I stop the loop 45 seconds before the termination and apply extensive local search in the best candidate to better approximate the local minimum (better explained below).

2.12 Parameter selection

One thing that is determined by the dataset is the mutation operator to be used (swap or inversion), described on the initialization section. All of the other parameters where essentially chosen by watching the performance of the large datasets, the smaller ones could be solved more easily.

I picked a population size of 1600 because this value offered enough explorative capabilities in the large dataset, while concurrently being able to run for a decent amount of generations. I set the amount of offspring equal to 2 times the population size. This is done because some of the ”randomness information” of the crossover operator is lost due to the fact that I apply the feasibility fix - therefore I thought that I should increase this part of exploration by having more offsprings.

I set the amount of times I apply the two opt repair mechanism for local search after mutation equal to 2. I did this because I did not want the local search in this part to be too exploitative and therefore I do not search for



(a) Tour 50 results. Generation 200 reached in 60 seconds

a larger amount of potentially better fixes.

I set the immigration factor (i.e. amount of candidates that are replaced) equal to 0.01 . This is because the randomness introduced might be a little too harsh.

I still needed to increase my explorative part though. This is why I chose a mutation rate of 0.3. This value seems too high, but empirically it managed to help me balance the exploration - exploitation tradeoff and achieve better results. The and k for selection and elimination respectively were again chosen with the performance on larger datasets as a criterion

2.13 Extensive Local Search

45 seconds before the end of the 5 minute time period, I stop the loop and pick the best candidate. Until the end of the algorithm, I consecutively apply mutation and repair the result using the 2 opt mechanism. If the result is better than the original, I replace the original.

This is a simple method of local optimisation to help us better approximate the local minimum and improve the result. This is due to the fact that excelling at approximating local minima is not something evolutionary algorithms excel in, generally speaking.

2.14 Code Performance Optimisation

Due to a variety of reasons (large datasets, missing edges, lack of symmetry etc) it is observed that the 5 minute period might not be enough to solve the problem with satisfying results. This is why code optimisation is really important - it essentially allows us to improve our results (e.g. allowing more generations or larger populations).

To improve the code, I used numba in some performance-critical functions (e.g. calculating the objective function, checking if a candidate is feasible, applying crossover and fixing infeasible candidates). However I believe I still have room for improvement and make the program more high performant.

3 Numerical experiments (target: 1.5 pages)

3.1 Metadata

More on parameter selection can be seen on (3.12). In the algorithms I use pop_size=1600, num_generations=800, num_parents=int(2*pop_size), alpha=0.995, mutation_probability=0.3, k=3 lso_probability = 1, lso_attempts = 2, immigration_factor=0.01. I apologise but this subsection is written on a hurry.

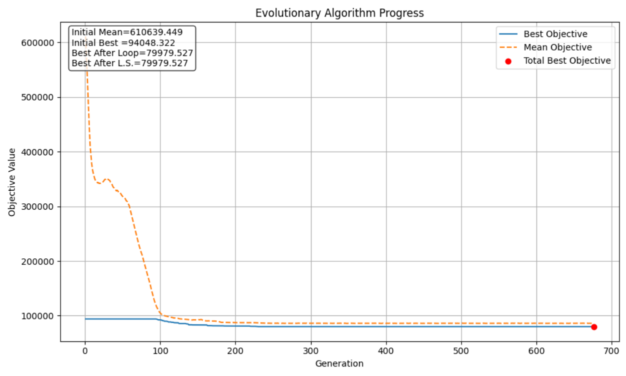
Python 3.8.10. 8GB Ram. AMD Ryzen 3 3250U with Radeon Graphics (2 physical cores 4 threads).

3.2 tour50.csv

The mutation type that the algorithm chooses is inversion. Optimal solution is (length=25795.193) : [25,12,27,28,36,23,44, 17,40,33,34,21,26,0,32,37,29,20,49,22,6,9,43,30,48,5,8,46,35,16,4,15,47,42,11,3,19,38,10,13,7,41,14,1,18,24,45,2,31,39,25].

The results asked can be seen on a legend on the corresponding figure.

Interpretation of results: The knn heuristic does a fairly good job at approximating the solution (at least compared to what we have later). This is why in this plot (and in all the plots to be honest) the best objective value per generation does not seem to change that much - the variability is much smaller than the mean objective which in the beginning comes from a diverse and fairly random population and in the end has converged to the final candidate.



(a) Tour 100 results, Generation 200 reached in 60 seconds

The convergence happens fairly early - this is due to the fact that the dataset is small and can be easily solved in a small amount of generations, especially when we integrate advanced techniques that are suited for the large dataset.

The reason why we observe that the the mean objective value has not become equal to the best objective value is immigration - in each generation an amount of random candidates forces diversity.

In terms of time the algorithm solves the problem quite fast. In terms of memory the dataset gives a much smaller space complexity.

It is worth noting that the mechanism of mass mutation cannot make any difference in the final candidate. This might imply the fact that the solution we get from the loop is already near the local optimum.

The result seems good, as it is better than the heuristic you posted and better than the one we had on the intermediate phase. Ofc evolutionary algorithms are not guaranteed to find the global optimum so I wouldn't say that the optimum we found here is global.

We should also note that there is significant overhead in the initialisation due to the fact the we use excessive local search to estimate the mutation type. This is not technically needed here, it is useful for the large datasets. Again, the algorithm is designed with the achievement of good results in the large datasets as a criterion.

In terms of the repetitive results, I observe that the algorithm really has a predictable character. There is a fairly small amount of different final optimal objective values we get. The total best I found is the one I present above. Most of them are on the [25800, 26900] range. The mean values on the otherhand follow the best ones with a small difference. This is due to the fact that I use immigration which forces diversity and variability in every generation.

I run the experiment 300 times and the histogram does not seem that informative, therefore I do not present it. The information that needs to be stated is the one above.

3.3 tour100.csv

The mutation type that the algorithm chooses is inversion.

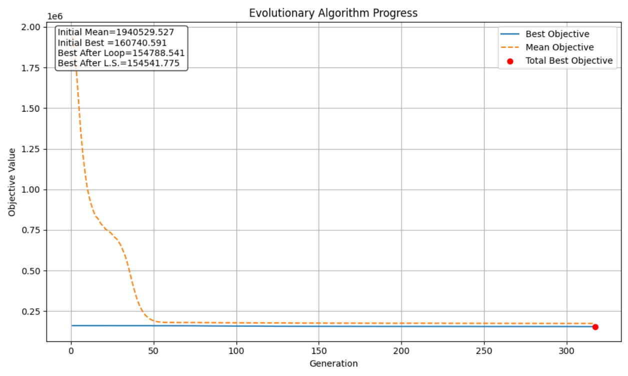
In essence the general interpretation of results in this problem is the same as the previous one. This problem is easily and quickly solved, the heuristic finds a good approximation, the final value is better than the heuristic you posted. I would say that by comparing my results with your heuristic the algorithm does better in this problem. This might be due to the fact that this problem, even though double in size in comparison with the previous one, has no disconnected edges. This might be the reason why the time needed for convergence is around the same - we do not have overhead associated with infeasibility fixing.

3.4 tour500.csv

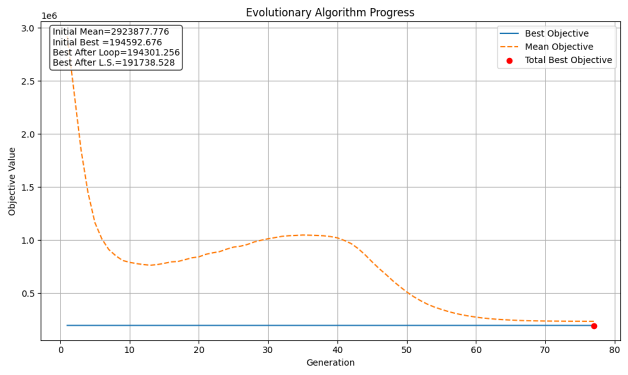
The mutation type chosen is inversion. In this case the interpretation changes significantly.

Now that we have a larger dataset we see that the algorithm stops at around epoch 350. This is due to the fact that the algorithm need more time to run on the larger dataset. I would say that the process is not finished, i.e. the results would be better if we continued the optimisation process. We can also compare the results with the posted heuristics and see that the performance is relatively not that much better (as compared with the other two problems).

We see that the mass mutation in the end managed to improve the results a little bit. This is due to the fact that the algorithm cannot go near the local optimum on its own on the time period given.



Tour 500 results



Tour 1000 results

Of course we could just reduce the population size to allow the algorithm to run for more generations. I observed however that the extra explorative capabilities that the dataset (1600 pop size) gives can make a difference in the final results. Maybe a more advanced diversity promotion mechanism could improve the explorative capabilities I need with fewer computational demands.

Another part that is problematic might be the process of infeasibility fixing. It is a process that greatly increases overhead. A better implementation might allow us to run for a larger amount of generations.

3.5 tour1000.csv

The mutation type that the algorithm chooses is swap

Again the results are equivalent (up to a point) with the previous dataset. Here we see that the algorithm really finds it difficult to run for a large amount of generations. Reducing the complexity by reducing exploration harms the results (as far as I observed).

Again, through my observations, I saw that the existence of the enriched population really helps us get better results. The mass mutation in the end has managed to really improve the results in this case - this is why I believe this is a technique that is worth having, even though it reduces the available time for the loop. This technique allows the premature algorithm to at least go near a local optimum that might need more generations to reach.

The algorithm really does not perform that well with a different choice of mutation type. Also, the extensive local search in the beginning can improve the quality of the initial population and the final result. Again this is something I observed through my experiments.

4 Critical reflection (target: 0.75 pages)

What are the three main strengths of evolutionary algorithms in your experience?

1. One strength of evolutionary algorithms is the fact that we can use the basic concepts that define the way these algorithms work (essentially the evolutionary algorithm block) to solve a vast amount of differing optimization problems, from neural network training to combinatorial optimisation (e.g. the TSP problem), without many constraints like availability of derivatives and so on. They are therefore easily generalizable.
2. One more strength is the fact that in essence evolutionary algorithms are methods for population control and utilization, and not simply optimization techniques (of course optimisation is a big part of evolution-

ary algorithms). For example, by proper use of the concept of evolutionary algorithms we can make the population resemble the Pareto front and solve the problem of Pareto front estimation.

3. Finally, evolutionary algorithms are effective techniques that achieve global optimisation. Compared to other global optimization techniques (e.g. random sampling or even random sampling + local search), evolutionary algorithms have ingrained in them elegant and effective techniques to help us balance the exploration - exploitation tradeoff and solve the problem in a better way than these other more simplistic methods.

What are the three main weak points of evolutionary algorithms in your experience?

1. Even though they can be used in a vast amount of different problems, evolutionary algorithms still demand care and attention so that they can actually excel and give good answers. We have to design many sub-algorithms (selection, variation, elimination, initialisation) and take into account how they will cooperate to give us the final loop. We might have to design advanced methods (e.g. local search, diversity promotion) to properly balance the exploration-exploitation tradeoff. The large amount of subalgorithms also come with a large amount of hyperparameters that have to be chosen.
2. Apart from the difficulty in design, evolutionary algorithms also are problematic in terms of time complexity. To make a comparison, we could design a random sampling + local search method more easily than the evolutionary algorithm. These methods are comparatively more easily parallelisable and in many instances can be considered as massively parallel - nowadays with the way high performance computing is evolving (GPUs etc) it could really make a difference.
3. According to my experience, evolutionary algorithms might suffer significantly by the curse of dimensionality. It might be quite demanding to design an algorithm that is able to give effective results for large problems. This problem is present in many techniques, I just think (with my limited experience) that it is more severe in E.A.s.

This project was my first serious attempt in designing and implementing an evolutionary algorithm. I experienced the process of trying to find (either come up with or research) methods to design the algorithm (appropriate mutation and crossover, tackling infeasibility, local search and diversity promotion). I also saw that it is demanding to combine all of these techniques effectively (e.g. understanding the needs of the algorithm and how to balance the exploitation - exploration tradeoff).

I also observed the need for high performant code, which essentially allows us to solve problems more effectively.

This problem is very appropriate for the use of evolutionary algorithms. It is not differentiable (combinatorial optimisation) which makes gradient methods unusable, and also no rigorous algorithm to actually solve it exists (for large datasets due to NP hard nature) - the choice of metaheuristics like evolutionary algorithms totally makes sense.

5 Further Improvement

In this section I will try to outline some ideas that should be implemented and tested to see how the algorithm will improve. It is obvious that there is still room for improvement, especially in larger datasets. I did not manage to implement these ideas due to time constraints - I already spent 50+ hours on the individual phase code and experiments.

First of all I should maybe try to check other techniques to fix the infeasibility. Maybe they would offer smaller overhead (which would allow me to run more generations on large datasets) or even better performance. The two opt repair seems quite naive to me, to be honest. The reason I chose it was because I wanted to integrate the two opt operator (taught in class) in my code to be able to apply local search too. I saw that it gave decent results but I did not manage to explore better options.

Another choice that gave me decent results and made me avoid entering the process of searching for a better alternative was the PMX recombination. I tried to implement the edge crossover but it did not give me the results I wanted - I am not satisfied with the amount of time I spent on this and I believe that I might be able to improve my recombination operator.

Maybe I should try more sophisticated diversity promotion mechanisms . I tried to implement fitness sharing but I could not make it work. A reason why it happened also has to do with the hyperparameter choice - it is a method that heavily depends on the choice of hyperparameters.

Of course, this problem would be solved if I managed to implement an algorithm to pick these parameters on its own. The idea of self-adaptivity seemed very interesting to me, when I first heard about it, but again the correct design of the algorithm and making sure that it works properly is something that demands attention and time.