Evolutionary Algorithms: Final report

Antonios Glioumpas (r0827913)

January 4, 2021

1 Metadata

- **Group members during group phase:** Wim Govers and Sichao Wen
- **Time spent on group phase:** 8 hours
- **Time spent on final code:** 42 hours
- **Time spent on final report:** 10 hours

2 Modifications since the group phase

2.1 Main improvements

Modification 1: The simple swap mutation operator was modified.

Modification 2: The 2-opt local search operator was implemented.

Modification 3: The main parameters of the algorithm were tuned to reach better solutions.

2.2 Issues resolved

Short description 1: The algorithm didn't explore the search space sufficiently and, as a consequence, it was always converging in a local minimum whose fitness value was always significantly higher than the provided optimal value estimation provided.

This issue was resolved mainly with the use of the local search operator (2-opt). The use of the operator in combination with the high mutation probabilities was making possible the "escape" from local minima.

Short description 2: The mutation operator was not providing enough variation to the offspring, especially for TSPs with a large number of cities.

This was solved by introducing the interval swapping mechanism and bigger mutation probability values which was proven useful especially for TSPs with bigger populations.

3 Final design of the evolutionary algorithm

3.1 Representation

The problem could be represented using the path representation and the adjacent representation. Both of these are permutations and a lot of known recombination and mutation operators exists for these representations. Finally, the the path representation was chosen. Every permutation of this representation is a valid solution for the TSP. The solutions to the problem are represented as sequences/orders of cities visited (1-dimension numpy arrays). Each of these sequences represents a route travelled by the salesperson.

3.2 Initialization

The population is initialized by generating random permutations. Since all individuals are generated randomly, the initialized population is expected to be diverse enough. Also, the size of the population is big enough to allow to this random generation of individuals produce enough diversity up to a size that does not slow down each iteration of the algorithm dramatically. In one of the experiments, the 2-opt operator was used on the entire initial population to improve fitness but the time cost was significant so finally this step was removed.

3.3 Selection operators

The selection operator that was chosen is K-tournament selection since it's simple to implement and a computationally cheap method (it doesn't need to evaluate the whole population). In one of the experiments, the k parameter was inserted in the individual solution's attributes in an attempt to use self-adaptivity, but since no significant improvement was observed, it was removed to favor simplicity.

3.4 Mutation operators

The mutation operator was modified in a way that instead of performing a simple swap mutation, it now swaps entire segments of an individual's genome. The length of the segments is controlled by an extra parameter (interval_length). With this variation of the swap mutation operator, the simple swap mutation (where only a pair of cities change place in the sequence) is only applied in case the segment's length (interval_length) is greater than the number of cities of an individual/candidate solution (ind_length) or the random selection of the limits of the two intervals-to-be-swapped overlap.
The probability of mutation of a new individual when the population is initialized, is given by alpha = 0.5 + 0.01 * np.random.randn() which corresponds to random samples from N(μ, σ^2) where $\mu = 50\%$, $\sigma = 1\%$ and a minimum value of 1%.

Furthermore, an extra mechanism was integrated in the mutation operator which reduces the probability of mutation as the number of generations/iterations increases. It depends on the values of num_of_iter and alpha_decr. The variable num_of_iter is just a counter of the generations and alpha_decr the value that multiplied by num_of_iter will define how much the probability of mutation will be reduced. The idea of using such a mechanism comes from the fact that in the early generations it is preferable to have a large probability of mutation (alpha) with larger intervals to swap but, as the time passes, both the probability and the intervals to be swapped need to be smaller so that the good solutions that have already emerged are not destroyed.

Self-adaptivity was used for the alpha value of an individual (probability of applying a mutation operator to an individual) When a new offspring is created through recombination, the alpha value of the child depends on the values of the parents and it's calcuated using the following formulas:

beta = 2 * random.random() - 0.5

alpha = parent1.alpha + beta * (parent2.alpha - parent1.alpha)

(random.random() generates a random float uniformly in the semi-open range [0.0, 1.0))

Note: The_interval size-reducing mechanism was neither implemented nor integrated to the algorithm since a constant value for the interval length was producing satisfactory results.

3.5 Recombination operators

The recombination operators considered were:

1. Partially Mapped Crossover,
2. Edge Crossover,
3. Order Crossover and
4. Cycle Crossover

The one used in this algorithm is "Order Crossover". It was chosen because this operator was designed for order-based permutation problems as the TSP, because of its simplicity concerning the implementation and by eliminating the other options considering their suitability in the TSP.

This operator can produce offspring that combine the best features from their parents in case one of the parents has a sequence of visited cities for which the total travelled distance is lower than the average distance of all the possible sequences that could be created out of the selected segment. If there is a little overlap between the parents, the operator rearranges the alleles in a way that the overlap gets decomposed with its parts placed in different positions in the offspring sequence. The recombination can be controlled by the segment_start and segment_stop variables. In this implementation, these positions are selected randomly. By introducing constraints into the length of the selected segment from one parent, similarities that (to an extreme level) can lead to an offspring being almost identical to one parent, can be avoided.

The self-adaptivity mechanism implemented into the algorithm is the recalculation of the alpha value of an individual solution (linked to the probability of a mutation taking place for an individual solution/sequence of cities). For each new iteration, the alpha value gets a new random value as described in the previous subsection.

3.6 Elimination operators

The elimination operator used in this algorithm is the $\lambda + \mu$-elimination. It is suitable for this problem since it keeps a decent diversity and esures that the average fitness of the new generation will be better that the previous one's. However, (λ, μ) selection could be a better option since it discards local minima and hinders self-adaptivity less than $(\lambda + \mu)$-elimination, but it was not finally implemented because other problems were prioritized to be solved before it.

3.7 Local search operators

The local search operator that was implemented is the 2-opt local search algorithm.

The improvement it brought to the overall result was remarkable since it could quickly reduced the amount of iterations needed to approach the given optimal values of the search space. After this point, the algorithm relied on the recombinations and mutations that took place in the population which produced new individuals from which the 2-opt operator could reach combinations of even better fitness values.

However, this stage of the algorithm is time-consuming as the number of cities increased, since the complexity of each round of the search is O(n^2), where n is the number of edges of the route map. Therefore, the operator was not applied to the entire population. Instead, in each iteration, the operator is improving only one random individual.

3.8 Diversity promotion mechanisms

The diversity promotion scheme that **was implemented but NOT used in the final version of the code** was Generalized Crowding.

For one step of a general crowding GA:

- Offspring generation (2 children from 2 parents),
- Distances are computed for all possible parent-child pairs,
- Matching parent-child pairs are computed, minimizing a distance metric.
  The complexity of a brute-force implementation of this matching is S! in the worst case, where S is the population size. For large S, where the size of S! becomes a concern, it is essential to avoid a brute-force approach. Instead, it is recommended to use an efficient algorithm such as the Hungarian weighted bipartite matching algorithm. This algorithm uses two partite sets, in our case the parents $\{p_1, ... , p_S\}$ and children $\{c_1, ... , c_S\}$, and performs matching in O(S^3) time. Even with this modification, the algorithms was slow for a large number of cities in the TSP and so, only a few iterations/generations could be completed resulting in a result far from the optimal when the time elapsed. **This was the main reason why the entire scheme was removed from the final version of the algorithm**.
- Tournaments are held by means of a replacement rule. The rule decides, for each matching parent-child pair, which individual wins and is placed in the new population. The selected rule in my experiment was the Deterministic Crowding Replacement Rule where the probability of a child replacing a parent is

$$p_c = p(c) = \begin{cases} 1, & \text{if fitness(child)} > \text{fitness(parent)} \\ \frac{1}{2}, & \text{if fitness(child)} = \text{fitness(parent)} \\ 0 & \text{if fitness(child)} < \text{fitness(parent)} \end{cases}$$

3.9 Stopping criterion

The implemented stopping criterion works as follows: if for a number of consecutive iterations (80 was chosen in the final version of the code), the best fitness values have a percentage of difference (maximum 0.00000001% difference was chosen in the final version of the code), the algorithm terminates. The EA could also stop once the fitness improvement remains under a threshold value for a given period of time (i.e., for a number of generations or fitness evaluations). Since not always a value close to the optimal one is provided, that is, a value that could be used as a threshold, it was deemed unsuitable not to use such a method. Alternatively, or in addition to the criterion above, after calculating the the population diversity and once it drops under a given threshold, the algorithm could terminate. However, such a calculation would be very computationally expensive, especially for a large number of cities, and so, it was not integrated.

3.10 The main loop

Algorithm 1: main loop, the **optimize** function

```
Initializations:
Read distance matrix from file and deduce the number of cities;
Initialize parameters : λ=150 , μ=150, k=4, alpha_decr=0.0001;
population ← list with λ random individuals;
current_iter ← 0  /* (counter measuring the number of iterations/generations)  */
;
prev_fit ← inf
convergence ← 0  /* (counter used in the convergence criterion)  */
;
while convergence < 80 do
    offspring = empty list  /* (initialization)  */
    ;
    for jj ← 0 to μ , step 1 do
        parent1 ← k-tournament_selection(population);
        parent2 ← k-tournament_selection(population);
        child ← Order_Crossover_operator(parent1, parent2);
        Add child in the offspring list;
        Apply the mutation_operator on the last child added in the offspring list;
    end
    Insert mutated clones of the parents into the offspring list (using the mutation operator);
    Apply 2-opt local search operator on the first individual of the offspring list;
    Apply elimination scheme: offspring list is sorted fitness-wise and the λ best individuals survive;
    fitnesses ← list with the fitness values of the surviving individuals of the population;
    best_fit ← the best fitness value of the current population;
    current_iter ← current_iter + 1
    if (prev_fit − best_fit)/best_fit < 0.00000001 then
        convergence ← convergence + 1;
    else
        convergence ← 0;
    end
    if timeLeft < 0 then
        break from the while loop;
    end
end
return best_fit
```

Note: only the essential parts of the algorithm are presented above. Lines used to visualize the results or to print them in the console are omitted.

The order with which the operators are applied on the population follow the scheme of Population Initialization followed by a loop of the stages: Selection of two parents, Recombination to produce one offspring, Mutation of the offspring. Once this process is finished, 2-opt local search is altering the genome of a random individual of the population which ensures the existence of an individual with a very competitive fitness value. The elimination scheme that follows indicates that this individual will survive to the next generation and will propagate its good characteristics to the offspring when it is selected as parent. Through all the experiments it was apparent that this technique vastly accelerates convergence to fitness values close to the global minimum. The main idea was that first the population is allowed to grow to various directions/combinations through the variation-generating mechanisms and then keep only the best individuals that emerge.

3.11 Parameter selection

The parameters were mainly chosen based on the results given by various experiments. The population size as well as the number of offspring had to be large enough so, (through the random initialization, recombination and mutations) the search space was sufficiently explored, and not too small so that a good candidate solution could not quickly take over the entire population. Since the group phase, we saw that for big λ values, the best fitness value improved, but the algorithm converged slower. The time limitation was a reason to be careful not to increase λ too much. Concerning μ, the fitness increased as μ increases till μ has the same value as λ. After that μ didn't affect performance a lot, unless it had an extreme value which was also slowing down each iteration.

Big K values were avoided since such a choice leads to a higher selection pressure from the K-tournament

selection operator, which could on average lead to even faster convergence when approaching the local minima and probably slightly worsen the final results.

The value of the probability of mutation is initialized to high random values. Additionally, a low decrement value is selected (alpha_decr = 0.1%) for every new iteration. Finally, the interval swapping mechanism, swaps a subsequence of 10 cities in the genome of an individual, in order to introduce a lot of diversity in the population. Such a choice was essential because the elimination scheme quickly reduced diversity and discarded candidates with high cost values that could potentially contain low cost subroutes in their genome.

4 Numerical experiments

4.1 Metadata

The parameters to be chosen in the current implementation are the following:

- λ = 150 (size of the population)
- μ = 150 (number of offspring in each iteration)
- k = 4 (parameter for the k-tournament selection)
- alpha, alpha = 0.5 + 0.01 * np.random.randn() which corresponds to random samples from N(μ, σ^2) where $\mu = 50\%$, $\sigma = 1\%$ and a minimum value of 1%
  when a random individual is created (Probability of applying a mutation operator to an individual)
- alpha_decr = 0.1% (the percentage drop of alpha for every evolutionary cycle)
- interval_length = 10 (number of cities in a sequence to be swapped inside an individual's genome)
- the convergence threshold = 80 (condition of the while loop inside the optimize function) as well as the convergence counter increment condition (prev_fit - best_fit) / best_fit < 0.00000001 (percentage of difference between two consecutive best fitness values of the population)

Main characteristics of the computer system on which the evolutionary algorithm run:

CPU Specs:

- 4 Cores,
- 8 Threads,
- Processor Base Frequency 1.60 GHz,
- Max Turbo Frequency 3.40 GHz

RAM: 8,00 GB (7,38 GB usable)

Python Version: 3.9.0

4.2 tour29.csv

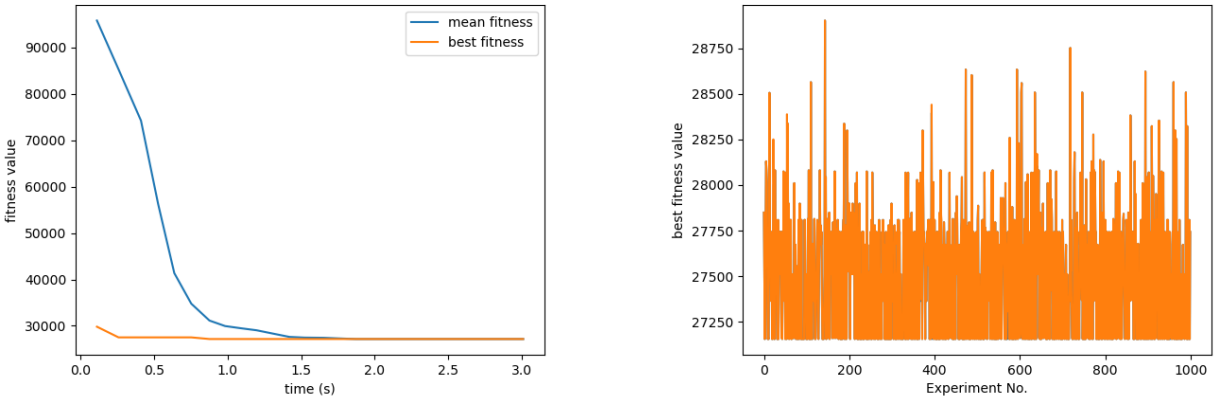

Figure 1: (Left) Example of a convergence graph using tour29.csv. (Right) Best fitness values from 1000 independent experiments using tour29.csv.

Best individual (sequence of cities): [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 4 7 3 2]

Fitness value of best individual : 27154.488399244645
number of iterations: 37 time needed for each iteration: 0.08 sec

The results illustrated in the graph showing the overall best fitness of the population at the end of each one of the 1000 separate experiments indicate that the values fluctuated between 27250 and 29000 with most of the values being closer to 27750. Comparing these results with the given optimal value of 27200, the conclusion drawn is that the algorithm performed well when it came to the final solution as well as how fast it reached this values. However, the convergence graph makes apparent that since the mean fitness value of the population quickly approached the best values, the diversity of the population dropped equally rapidly. Finally, the small number of cities in this example (29 cities) cannot be a representative value of how well the algorithm copes with memory usage.
Concerning the parameters, the only difference between the values mentioned in subsection 4.1, was the convergence threshold (condition of the while loop) that in this experiment was set to 20.

4.3 tour100.csv

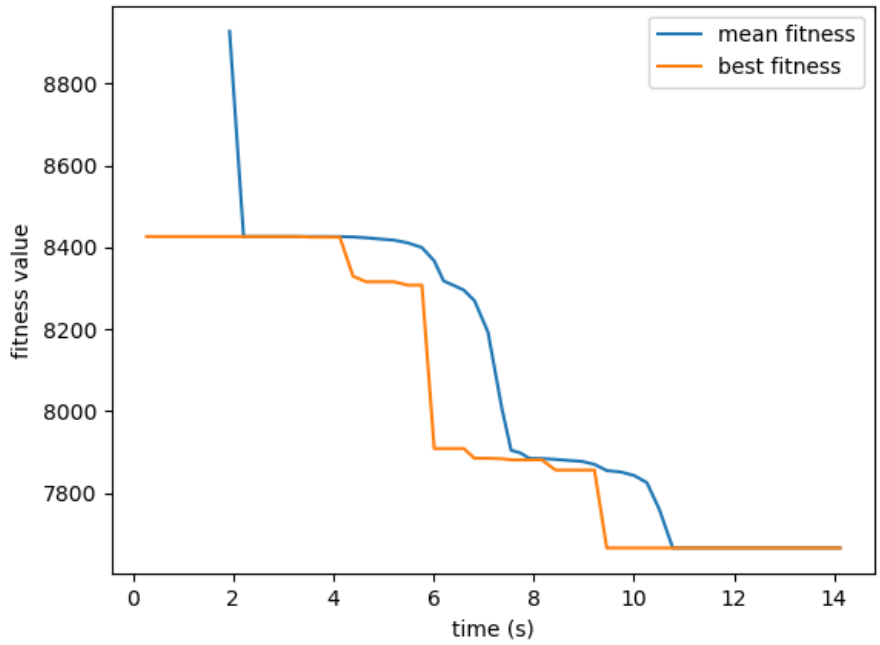

Figure 2: Example of a convergence graph using tour100.csv.

Best individual (sequence of cities): [31 82 21 75 99 81 92 52 60 2 47 35 25 71 55 80 16 20 63 24 97 98 48 40 44 78 33 90 85 96 70 27 22 77 69 1 29 34 17 95 76 23 68 18 59 49 5 56 57 19 73 39 32 74 38 13 8 86 4 51 54 41 12 89 61 43 58 67 42 66 87 83 0 50 65 15 79 72 88 64 93 14 6 84 36 94 46 91 28 10 37 45 7 62 53 11 30 3 9 26]

Fitness value of best individual : 7665.958441403743
number of iterations : 56

time needed for each iteration: 0.25 sec

In this experiment, the final solution came close to the approximation of the optimal value which is estimated to be around 7350. In Figure 2, the plateaus where the best fitness value is stable indicate that for several generations a surviving individual outmatched every other candidate solution and potentially took over the population until new offspring with better fitness values were generated through recombinations and mutations.

Concerning the parameters, the only difference between the values mentioned in subsection 4.1, was the convergence threshold (condition of the while loop) that in this experiment was set to 20.

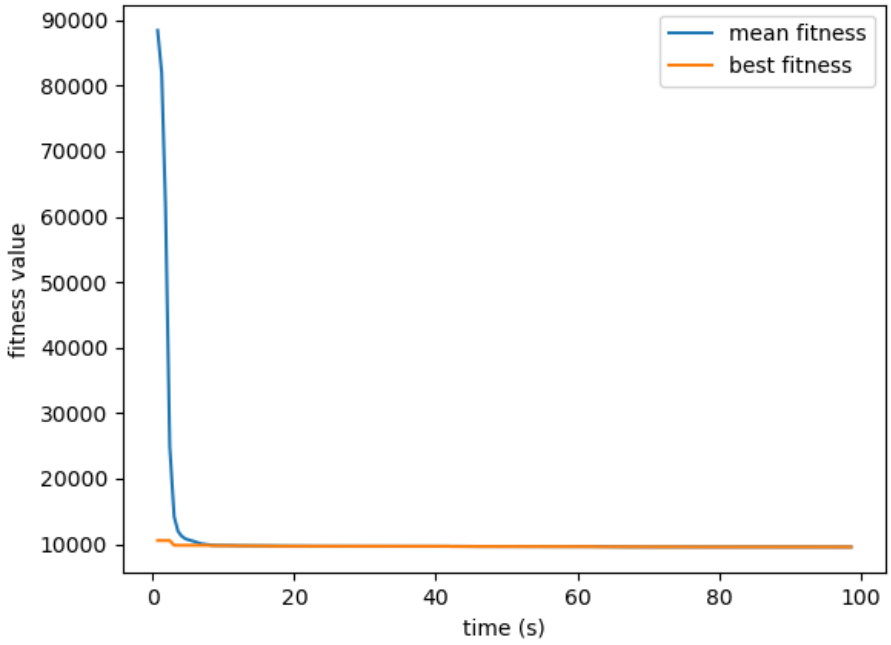

Figure 3: Example of a convergence graph using tour194.csv.

4.4 tour194.csv

Best individual (sequence of cities): [131 129 126 124 125 118 121 117 107 106 104 105 102 101 108 112 113 110 103 100 98 93 88 89 97 84 85 64 19 62 35 58 61 81 79 86 75 70 24 22 12 15 7 5 0 1 2 3 6 10 13 16 25 23 20 17 32 27 28 21 26 11 8 9 4 14 18 29 31 30 34 37 40 43 41 49 48 54 53 45 47 51 52 55 57 50 46 42 39 33 38 36 44 56 59 68 73 71 74 77 90 92 95 94 96 91 87 82 80 78 76 69 63 67 65 60 66 72 83 99 109 111 114 115 116 120 119 122 123 127 132 134 128 130 135 142 147 159 165 170 184 192 187 188 190 191 189 193 181 175 168 162 160 163 171 178 185 186 182 173 172 174 183 176 180 177 179 169 167 164 166 161 158 157 154 150 146 151 140 152 156 153 149 143 138 137 141 145 148 155 144 139 136 133]

Fitness value of best individual : 9572.053829685921
number of iterations : 237 time needed for each iteration: 0.42 sec

The 2-opt local search is producing really competitive individuals that drop the average fitness value of the population quickly. Then, these high-performing individuals are competing with each other producing slightly better solutions for every new iteration. In the graph, because of the scale of the vertical axis, this competition is not clearly visible since the final values are really close to each other. The final result is close to the optimal value (approximately 9000).

Concerning the parameters, the only difference between the values mentioned in subsection 4.1, was the convergence threshold (condition of the while loop) that in this experiment was set to 50.

4.5 tour929.csv

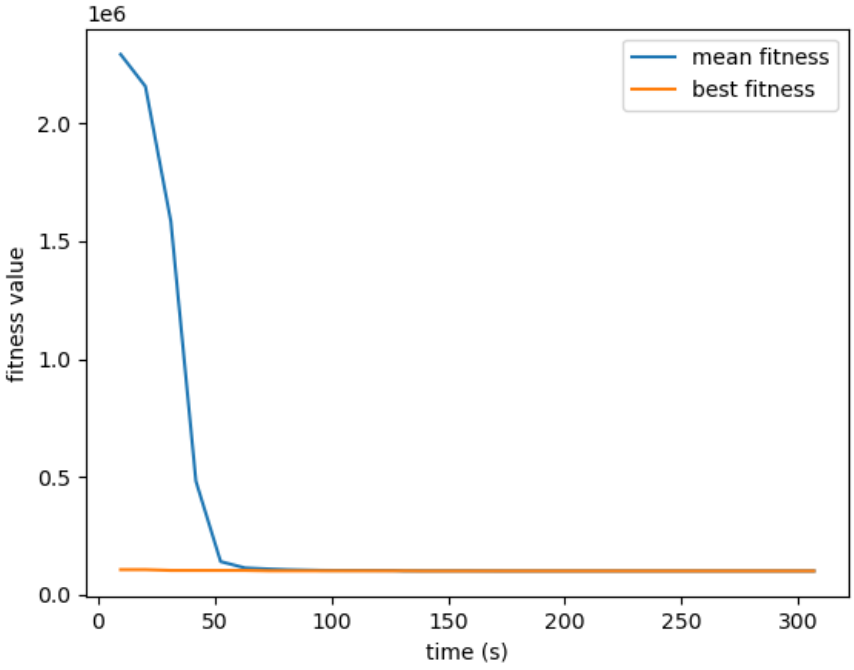

Figure 4: Example of a convergence graph using tour929.csv.

Fitness value of best individual : 101306.0799574213
number of iterations: 55
time needed for each iteration: 5.45 sec

In this experiment, it was clear that in each iteration, the most time consuming part was the 2-opt operator as the big size of an individual's genome (sequence of cities) increased the number of potential edges to be swapped to create new neighboring cities. Moreover, 300 seconds were not enough for the convergence criterion to be met. Similarly with the rest of the experiments, the population quickly comprised of individuals of values around a minimum whose value is close to the optimal value (approximately 95300). This conclusion is drawn out of the fact that the mean fitness value of the population is different but close to the best fitness value. The more time is given to the algorithm, the chances of a beneficial mutation increases that can allow the 2-opt operator to further improve fitness.

5 Critical reflection

Through this course I was able to understand how computer science can solve computationally intense problems by applying nature's evolutionary strategies that produce species able to adapt to the challenges of their environment.

It is apparent that evolutionary algorithms can easily be adjusted to the problem at hand. Almost any aspect of the algorithm may be changed and customized. Furthermore, since multiple offspring in a population act like independent agents, the population (or any subgroup) can explore the search space in many directions simultaneously. This feature makes it ideal to parallelize the algorithm for implementation even though it was not the aim of this project.

However, the freedom that parametrization offers to applying genetic algorithms to computationally demanding problems, determining suitable values to these parameters can be a problem by itself to solve. Any inappropriate choice will make it difficult for the algorithm to converge or it can simply produce meaningless results. The complexity of the parameters' selection can be limited by avoiding hyperparametrization and by trying to define the minimum amount of parameters to tune. Self-adaptivity can also be very useful so some of the parameters values are determined by themselves through "natural selection".

For the Travelling Salesperson Problem, where the main objective function is simple to calculate, so the individuals have to satisfy one fitness criterion (that of a low route traversing cost), it is quite straightforward to solve it using an evolutionary algorithm.

I was surprised by the similarities between the techniques used and how much inspiration can be drawn

from the ways nature works. Seeing how hard problems can be solved, even with approximations of the optimal values, in a fraction of the time that would be needed with a brute-force approach, I would definitely consider applying evolutionary techniques to problems I might encounter in the future both in my academic and professional life.