Evolutionary Algorithms: Final report

Rajat Sharma (r0846712)

January 3, 2021

1 Metadata

- **Group members during group phase:** Stijn Staring and Pieter-Jan Vrielynck
- **Time spent on group phase:** 15 hours
- **Time spent on final code:** 360 hours
- **Time spent on final report:** 30 hours

2 Modifications since the group phase

2.1 Main improvements

Many changes were implemented for the individual phase compared to the group phase. The changes can be broadly categorised in two parts. The first part caters to the new things that were introduced to the code of the group phase and the other type of changes were aimed at improving the group phase code. Improving was at multiple fronts - from the convergence to the quality of the solution. Following are the new additions to the code, each accompanied with a description explaining what they were. Elaborate details follow in next section.

Selection: Three new selection schemes were coded. They were: Top-k, Roulette Wheel and Sigma Scale. Roulette Wheel and Sigma scale involved calculating the probabilities of each individual to be selected. In other words,the last two techniques come under "fitness proportionate selection".

Crossover: Two additional crossover techniques were chosen based on research. They were respectively Edge and Order Crossover. Edge recombination is quite a well-known technique for TSP problems which is basically a more refined version of the alternating edges crossover. The offspring is expected to inherit the maximum number of edges from the parents and accordingly, randomised edges should be minimized. This is what the edge crossover tries to achieves. It uses an *edge map* which comprises of the list of the edges that are incident to each other in the parent tours and are known to lead to locations that are not yet part of the offspring. Such edges are called active edges as they can be utilised for the extension of the tour. Thus, the target here becomes to extend the tour by the addition of a city with the minimum number of active edges. Order crossover, on the other, allows two points to be chosen randomly on the parent tours. The part between the chosen cut points is added to the offspring and the remainder of the locations in the offspring are filled by adding the cities from the second parent starting from the second cut point. For order crossover, it is possible to have two children per recombination depending on whether first or the second parent was chosen in the first step.

Mutation: Two additional mutation operators were added, again based on the literature review. They were namely insert and swap mutation. Insert mutation involves choosing two cities at random and then shifting the second one in order to bring it next to the first one. Swap mutation, as the name says, consists of swapping two cities selected randomly from a given tour.

Elimination: *Lambda-Mu* elimination strategy was coded. It returns the new population constructed/formed only out of the new offspring population and discards the old population entirely.

Stopping Criteria: An improvement was made to the existing termination condition and another method for stopping the iterations was added. The first change was made to give more control to the user over the stopping criteria. The changes made here were that the user could control the number of iterations that needed to be taken into account while calculating the difference between the mean of best fitness and mean fitness. This value when went below a certain threshold (again user-defined) would break the loop. The other new stopping condition added broke the loop when the mean fitness became equal to the best fitness.

Final 2-Opt Optimization: Genetic algorithm explores the solution space and then with time and of course the parameters set for the algorithm starts to converge to an optimal solution. In an ideal scenario without any time constraints, it would have been possible to run the algorithm for a large number of iterations, explore the search space even more properly and then finally obtain a good convergence (although not always the case, depends on the way algorithm has been programmed). As the code had to run for a maximum of 5 minutes, it was necessary to not only explore but also to exploit. To achieve this, exploitation was done on the best solution of the genetic algorithm (the converged solution). The technique employed was a two-opt local search operator.

Additional Methods Implemented: These techniques were programmed in the algorithm at my end after referring to a large number of research papers. Many researchers have employed various kinds of techniques in their algorithm to improve the solution of the TSP problem and based on this literature survey, I understood, analyzed and then implemented few of these techniques to improve my solution. They were found to improve the overall solution in almost all the instances.They are as follows:

- **Self-Adaptivity** - This associates every candidate solution with its own mutation probability which is modified in every iteration during the recombination step
- **Elitism** - Top individuals from every population are automatically promoted to the next generation (population). Their number is user-defined and the algorithm takes this number into account for calculating the number of children/offspring desired
- **Variable Mutation Rate** - Considering the different kinds of tours, one common constant mutation rate is not really an ideal way to implement mutation. Thus, based on previous research, variable mutation rates were introduced for each of the tour. In the code, tour 29 and tour 100 had one mutation rate function whereas the tours 194 and 929 had another. The mutation function for the first two tours is dependent on the number of iterations and the mutation rate decreases with iterations. In the second case, the mutation rate is a function of the minimum route length and the mean route length of the population for each iteration and as the solution converges, the mutation rate increases. This is recommended for larger tours because it is necessary to maintain sufficient diversity in the population to reach the optimal solution in good time. Too slow convergence is also not desired. Thus, for tours 194 and 929, the mutation rate increases gradually with convergence ensuring good diversity in every generation.
- **Advance Crossover** - This approach too has been employed by a lot of researchers. In this, during the recombination phase, if the offspring produced from two parents has a better fitness than the worst of the parents, it is chosen to replace the worst parent. The crossover is then carried out between this new *parent* (the offspring) and the other parent. This leads to even better offspring as the set of new parents is better than the set of old parents. The only disadvantage are the extra calculations involved and thus this is computationally expensive and this disadvantage becomes even prominent in the cases where the tour length is very large.

Plots: Visualization of the iterations/convergence is a great way to quickly analyse, assess and understand the whole algorithm and how the process evolves. Thus the package of *Matplotlib* was used to automatically plot 8 plots overall - first four were against the number of iterations and the remaining 4 were against time. The 4 types of plots were - mean fitness, best fitness, optimization and a combination of all three together for a holistic view. The plots includes the two main phases - first the iterative process of genetic algorithm and secondly, the optimization part where the best converged solution was further exploited using the 2-opt local search multiple times.

Experimentation: As the number of parameters in the genetic algorithm grow, the possibilities and the variations too increase quickly.The total number of combinations that one can experiment with drastically shoot up. Playing with all of them requires a lot of time. However, that said, they also present an opportunity to understand the algorithm from a closer perspective. This is precisely what was done. An attempt was made to experiment with some parameters and their values. It took a lot of time but in the end, one acquired a solid understanding of the whole process. Some experiments that were tried out:

- **Application point of 2-opt local search operator** - The 2-opt was applied on the whole population, the two parents, the offspring produced after recombination and the convergence rate as well as the final solution obtained were compared. Results discussed in later section.
- **Mutation rate**s - Constant, increasing and decreasing (with iterations) mutation rates were tried and again the results were compared. It was only after this study that it decided to employ two different mutation rate functions for different tour lengths as it was evident that having a uniform common rate of mutation is ineffective in maintaining adequate diversity and variation in each generation.
- **Recombination Rate** - Recombination with a crossover rate (beta) was also experimented with values ranging from 0.1 to 1 and the results obtained coupled with the research papers proved that recombination is

best when it is equal to 1 and thus even though the code has the option of incorporating beta, its value is set to 1 for all the tour lengths.

- **Two Offspring Order Crossover** - As mentioned previously, this crossover has the possibility to produce two offspring for each recombination. Experiments were done to check the impact on the offspring population and overall solution if two offspring were produced for each recombination in every iteration and the results are analysed later in the report.
- **Calibration of the parameters** - Lot of calibration and tuning was performed to gain insight into the change in behaviour of the genetic algorithm as the values of multiple parameters were changed. Based on such experimentation only, the final optimal values for each tour length were derived and stored for the testing purpose (*tourValues* variable/dictionary in the code). Due to time constraints, definitely not all possible combinations could be studied, however a good understanding was achieved overall about the genetic algorithm.

2.2 Issues resolved

The code from the group phase was basically made more advanced by adding more strategies for the different operators which directly influenced the quality of the results. By quality, it is meant that the convergence rate increased as well as the solution became closer to the optimal for the specific tour under consideration. This was, in other words, the issue of the time-accuracy trade-off which was a problem that was present during the group phase. Another problem that was existent during the group phase was the issue of the algorithm getting stuck in a local optimum which led to not-so-optimal solution. Furthermore, due to time-shortage, the parameters could not be properly tested and tuned and thus the values used for them during the group phase were not quite optimal. As was described in the previous section, new operators for selection, crossover, mutation, elimination were introduced along with some advanced strategies for crossover coupled with local optimisation (issue 1: local search operator) for further improving the quality (issue 2: convergence rate and accuracy) of the solution. Other things such as stopping criteria, plots for visualisation were also incorporated in the code during the individual phase. Calibration studies for the involved parameters were extensively carried out and most optimal value for each parameter was found and set for each tour length. The obtained results show that the solution for each tour length did improve, in terms of convergence rate and being optimal, thereby mitigating the main issues.

3 Final design of the evolutionary algorithm

3.1 Representation

Candidates for the TSP problem were represented using an array (Python List) of integers. Every city was given a number starting with 0. Due to the fact that the salesman had to return to the same city he had started with (here 0), the starting and the end cities in each of the individual was the same (i.e. 0). Thus, a list of integers starting with 0 and going up to *number of cities - 1* and the final city again being 0 (returning to it) represented a single candidate for a given tour. This is literally the easiest way to represent a candidate solution and is the most preferred way considering that it is relatively easier to perform numerical and list operations. Accordingly it was chosen.

3.2 Initialization

The population was initialized with randomly generated individuals for a fixed size of the population which was user-defined. This size was mainly determined based on the type of tour (larger the tour, smaller the population size to account for the 5 minute time constraint). A good idea about the size was also deduced on the basis of multiple trial runs (experimentation) and by observing the results of these runs. The candidates (list of integers) were also randomly generated. One check was implemented during the generation of population prevent repeated candidates. Experiments were conducted by applying 2-opt local search on the population so as to have good candidates right at the beginning, however this proved to be highly slow and computationally expensive and thus was discarded. To sum up, the candidates were randomly generated and consequently the population was also randomized. However, this is one point that could not addressed completely due to time constraints. Having expanded my code in other ways to accommodate different types of operators and other techniques, it did not leave me with enough time to tackle this issue of lack of diversity in the initial population. Despite this, an attempt was still made to implement *crowding*, however even after going through DeJong's paper, its implementation was not clear and thus, this fact coupled with the paucity of time made it quite difficult to optimize the initial population. To be able to have a good scheme for initializing the population is certainly a great way to improve this code further. Still, to account for the diversity, other techniques such as variable mutation rate, self-adaptivity etc. were adopted.

3.3 Selection operators

Referring to the previous section, four selection operators were implemented and which one was chosen for a specific tour was dependent on the user. They were chosen on the basis of literature review of a large number of research papers and books. Some specific techniques are known to be effective in producing good solutions (for e.g. k-tournament is quite popular) and thus were considered. There exist very sophisticated schemes however producing solution within a reasonable time limit and its ease of implementation were also important criteria and therefore, going for a highly advanced scheme would have been very complex with no guarantee that the solution would be obtained within a time limit of 5 minutes. For k-tournament and top-k, the value of k needs to be chosen and set initially. The results obtained based on one of these four selection criteria were found to be quite satisfactory.

3.4 Mutation operators

Four mutation operators were used in total. Swap and insert mutation operators have been described previously. The other two were scramble and inverse mutation. In scramble mutation, the segment between randomly chosen points in the candidate is scrambled/shuffled. For the case of inverse mutation again random points are chosen and then the points between these two cut points are inverted/reversed. From these four operators, scramble and inverse operators produce the maximum mutation though based on the past research, inverse mutation is the most preferred and this fact was verified during trial runs too as the results were better than with the scramble operator. The reason why inverse wins over scramble is that it is possible after scrambling for the final scrambled candidate to be the same as the original one (no mutation actually) whereas with inverse, there is guaranteed mutation. Swap is expected to produce least mutation and insert mutation is also not that strong. Thus, based on this understanding and calibration studies, it is natural to see that for smaller tours, too much mutation would be problematic whereas too less mutation for larger tours would be ineffective. Hence, different tours had different mutation operators. None of the mutation operators require special parameters, just the route and mutation rate are necessary. Yes, both self-adaptivity and variable mutation rates were implemented. The user has the option to enable self-adaptivity and then each candidate gets a randomly generated mutation rate between 0.3 and 0.39. When it was disabled, variable mutation rate becomes enabled. Mutation is one of the most influential parameters and thus significant differences were observed with various types of mutating schemes as well as with varying mutation rates. For the final code, carefully chosen values/operators were chosen for the four different tours and used.

3.5 Recombination operators

Three recombination operators were implemented. Order and Edge crossover have been described previously. The third one was the sequential crossover. Given two parents, an offspring is produced by a sequential crossover using good edges (which could be present in the parents or not) where their goodness is determined on the basis of their values present in the parents' structure. Because this operator is not a function of the parents' structure, it can also produce new, better edges that are not present in either of the parents. Thus, the chances of producing good and better offspring increases a lot compared to the others. The choice of these three crossover operators was done on the basis of previous research and ease of implementation, although implementing even these three was very tricky and challenging. Sequential operator especially was known to be very good at producing high quality offspring that eventually led to faster convergence and better solutions. This is precisely why it was chosen for every tour in the submitted code. Furthermore, the chances of not having good parents is reduced due to good selection schemes that choose based on the fitness of the candidates such as Roulette and Sigma Scale. Also, to avoid the situation where the parents might not be that good, an advance crossover technique was used which was explained earlier. Thus, overall, combining the selection and the advance crossover schemes with such good recombination operator (sequential) mitigates the problem of producing bad offspring to quite an extent. None of the crossover operators have special parameters. On comparing the three operators through rigorous experimentation and calibration, it was inferred that in most cases, order was the fastest, followed by sequential and edge being the slowest whereas in terms of the quality of the solution and convergence rate, sequential leads the pack followed by edge and order. To have a balance, sequential was chosen for all the tours. Also, right after the recombination, mutation took place based on either self-adaptivity or variable mutation rate. This further ensured convergence to the most optimal solution, in other words, rejection of the poor candidate. As mentioned couple of times, due to a 5-minute limit and overall lack of time and resources, understanding, coding complex schemes (arity ¿ 2 etc.) and their application was kept to a minimum. A balance was sought between all operators, methods and techniques, keeping in mind all the constraints, and a best shot was taken to achieve the objective of producing a workable and a satisfactory solution to the TSP problem.

3.6 Elimination operators

Two methods for eliminating were deployed in the Python genetic algorithm code - *lambda + mu and lambda-mu*. As such, both of the techniques do not require any special parameters. Only the offspring generated and population size are required, which are readily available. These two methods are again very popular and ubiquitous

in several research papers. Based on the positive feedback about these operators from these papers, it was decided to choose them and apply them in the code. They were also not that convoluted to implement. The results obtained act as evidence in verifying and validating the satisfactory effectiveness of these elimination operators.

3.7 Local search operators

Two-opt local search optimization was used. Previous section describes how this particular scheme was used at various points in the code to check the effect on results. Furthermore, an attempt was also made to implement advance versions of this k-opt local search. First was the 3-opt and the other was the Lin-Kernighan heuristic (LKH) which is pretty much the best heuristic available till date for TSP problem. However, on referring to the past research, it was concluded that the benefit from 3-opt wasn't that much and it proved to be quite expensive. Usually above two-opt, it becomes computationally intensive. With LKH, after each iteration the value of k is changed rather than having it fixed for all the iterations. This is very a good heuristic and leads to very optimal solutions, however implementing it in itself is a totally different research project. Considering the time and resources available, it was not feasible to code and apply the LKH to TSP. This is definitely one of the best improvement that can be done to the existing code. Thus, I settled on 2-opt for this project and after finally experimenting with various points of application of 2-opt LSO, it was decided to implement it on the best solution obtained after the application of the genetic algorithm (converged solution). It was then again applied on the resulting optimized solution and this process was continued for a fixed amount of time. Each 2-opt optimization ran for a specific time keeping in mind the 5 minute time limit. Both of these variables were user-defined i.e. number of 2-opt iterations and the time for each 2-opt and were set up before the start of the algorithm. The results obtained after this optimization which is essentially exploitation of the converged solution was found to result in very good solutions, sometimes improving the converged solution by around 2000-2500. Thus, this 2-opt exploitation technique was very favourable to the algorithm and was extensively used leading to a good, optimal and close-to-optimal solution.

3.8 Diversity promotion mechanisms

The methodology for incorporating diversity in the algorithm was accomplished in the form of various operators for selection, crossover, mutation and elimination. Maintaining diversity is essential to the success of the algorithm and it also plays a pivotal role in the convergence and in producing a good solution. All these different forms have been explained in the previous sections. Where ever extra inputs were required for these methods, they have been delineated and described appropriately. While implementing methods such as fitness proportionate selection, advance crossover, variable mutation rate and self-adaptivity, diversity promotion was certainly kept in mind and their application does prove to significantly improve the obtained solution in terms of convergence and being closer-to-the optimum. Special schemes do exist, however developing (coding) them might not be so straightforward. One of the major goals behind developing the code for the TSP problem was to ensure that a working and sufficiently optimal solution (convergence) was achieved under the time limit of 5 minutes. With this objective in mind, all other developments related to the code were planned and undertaken. The solutions produced for various tours using the submitted Python code are quite within the range of being called a good satisfactory solution. They validate and verify a successful and an effective implementation of such diversity inducing and maintaining methods in the genetic algorithm for the TSP problem.

3.9 Stopping criterion

From previous section, two stopping criteria were used. Both were used in conjunction with each other in the code. First one was the keeping track of the difference between the mean of the best fitness and mean fitness for a specific number of iterations unless it went below a certain threshold and the loop broke. Both these values - number of iterations to be considered from the history and the threshold - were coded in as variables that were required at the start of the code. However, due to good calibration of the parameter values for each of the tours, these criteria did not come into action that often. Convergence was always almost achieved even before this termination criteria was met. Due to tuned parameter values, for all the tours, the 5 minute limit was always respected. Nonetheless, it was a good check to have just in case the number of iterations were set to be large and convergence was achieved quite early. In this case, the code would stop due to this criteria being met leading to time and computation efforts/cost savings.

3.10 The main loop

The main loop of the genetic algorithm requires several parameters to be set (17 in total). Some of them are refined and tuned for a specific tour and the rest are fixed and common for all the tours. Thereafter, the conventional two *for* loops run - with the outer loop handling the number of iterations while the inner loop deals with the production of the children. Then based on the choices, the four major operators in the order - selection, crossover, mutation and elimination are used with each operator choosing from one of its options based on the user input.

The order is usually the standard and conventional approach for the TSP problem. This loop is simultaneously being timed. For each iteration, two values are displayed- the mean fitness of the population and the best fitness. Depending on whether the iterations get over or the termination criteria is met, either way, after the main loop, the 2-opt LSO starts (is applied on the best solution and for multiple times on each optimized solution) for a certain time (again user controlled). Once that finishes (within the time limit of 5 minutes), the plots are drawn. Towards the end, a summary of the final results consisting of the route length after GA, best route length, best route (best meaning after 2-opt LSO) and total time taken by the main loop as well as the complete algorithm are calculated and displayed in the console for the user. The offspring are generated one-by-one, followed by mutation and elimination. For both the selection and elimination, the operation is done without replacement. This is the main loop of the genetic algorithm in a nutshell.

3.11 Parameter selection

This point has been addressed in detail in the previous section and also later in the report. Extensive and elaborate experimentation was done which involved calibrating various parameters and conducting trial runs. Based on that, an deep understanding of how each parameter (and also in conjunction with other parameters) influenced the results and what role they played was acquired. This took a lot of time but was worth the effort and it helped me get good values for the final code submission. Additionally, it was planned to conduct a Sensitivity Analysis which determines the scope of influence of each involved parameter, however, it is not easy to perform. Sensitivity analysis/study requires much more time and this was unfortunately not possible as implementing (reading, understanding, coding and testing other techniques, methods, approaches, schemes) consumed time. But this study can give a deeper insight into how parameters play a role (if) in the genetic algorithm.

3.12 Other considerations

Yes, elitism was coded and employed in the genetic algorithm. The total number of the elite candidates that pass to the next generation directly was a user-defined parameter that needed to be set in advance. The minimum value could be 0 meaning that elitism was "inactive". The children variable was calculated accordingly (*number of children - eliteRoutes*). Elitism has an impact on the final solution however it was a bit difficult to ascertain something concrete regarding the number of elite candidates required. For sure, it is not possible to have a fixed common value of elite candidates for all the tours. Multi-threading was considered, however but due to paucity of time, it was not feasible to refactor the whole code to have processes on different threads. But as the code ran fine within a reasonable time frame (not too slow) for all the tours, it was not deemed "very necessary" and preference and time was given to implementing other techniques and schemes to improve and refine the solution and code.

4 Numerical experiments

4.1 Metadata

In total, there are 17 parameters that control the flow and the processing of the genetic algorithm. They need to be set up prior to the running of the algorithm. For the final submitted code, 11 of these parameters were preset after a lot of experimentation. They, in my view, produce the best possible solution under the time constraint of 5 minutes. Table 1 gives a good overview of the parameters. Some important points with respect to Table 1.

1. For the selection, crossover, mutation, elimination operators, the values represent different kinds of techniques for that particular operator
    - *k Tournament*: 1, *top-k*: 2, *roulette*: 3, *sigma Scale*: 4
    - *Order*: 1, *Edge*: 2, *Sequential*: 3
    - *Scramble*: 1, *Inverse*: 2, *Insert*: 3, *Swap*: 4
    - *lambdaPlusMu*: 1, *lambdaMu*: 2
2. For *advanceCrossOverOp* and *selfAdapt*, the two values, 0 or 1, basically denote whether they are enabled (1) or disabled (0). For *selfAdapt*, the function *routeMutateRate* calculates the mutation rate for each individual (refer to code for more details)

Regarding the technical specifications of the machine on which the genetic algorithm was coded and run and the Python version:

- Apple MacBook Pro [2014 Model]
- Processor: 2.8 GHz Dual-Core Intel Core i5
- Memory 8 GB 1600 MHz DDR3

- Python: 3.8.5

S.No.	Name	Description	Type	Value(s) [fixed value]
1.	selectionOp	selection op.	preset	1 or 2 or 3 or 4
2.	crossOverop	crossover op.	preset	1 or 2 or 3
3.	mutationOp	mutation op.	preset	1 or 2 or 3 or 4
4.	eliminationOp	elimination op.	preset	1 or 2
5.	advanceCrossoverOp	advance crossover op.	preset	0 or 1
6.	eliteRoutes	no. of elite candidates	preset	int (≥ 0)
7.	selfAdapt	self-adaptivity	preset	0 or 1
8.	sizeOfPopulation	population size	preset	int (> 0)
9.	children	children size	preset	int (> 0) - eliteRoutes
10.	iters	iterations	preset	int (≥ 0)
11.	numberOfTwoOpt	number of 2-opt	preset	int (≥ 0)
12.	k	k-tournament, top-k	fixed	int (> 0)[3]
13.	alpha	initial mutation rate	fixed	float (]0, 1]) [0.5]
14.	beta	crossover rate	fixed	float (]0, 1]) [1]
15.	optiRunTime	time limit for 2-opt	fixed	float (> 0) [3 seconds]
16.	noHistoryIters	no. of previous iters. for termination	fixed	int (≥ 0) [5]
17.	tolerance	tolerance threshold for termination	fixed	float (]0, 1])

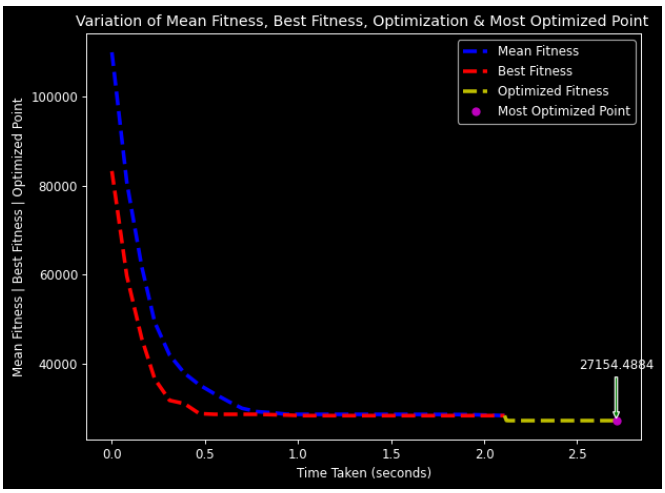
Table 1: Parameters for Genetic Algorithm

For the next section discussing about the results of the individual tours, I would like to mention that the plots shown and the description given for each tour is based on the results that were produced using the values (variable *tourValues* in the Python code) used in the final submitted code. As discussed previously, these values were arrived at by detailed calibration of the parameters for the complete genetic algorithm (including 2-opt local search) to finish within the given time constraint of 5 minutes.

4.2 tour29.csv

For this tour, the algorithm ran very fast (Figure 1a and figure 1b).The best tour length (27154.4884) was better than even the optimal value of 27200. This was made possible due to extensive calibration of the parameters. The figures prove that the set parameters values were quite optimal. In fact, this result was even replicated many times by choosing other options for various operators. This further proved the reproducibility of the results and the correct working of the algorithm with different settings and also highlighted the importance of calibration for finding such optimized parameter values.

The figures below show the plots illustrating the relevant data and as well a summary of the results that were obtained on running the genetic algorithm for the tour length of 29 cities.



(a) Convergence: Mean Fitness, Best Fitness, 2-Opt LSO

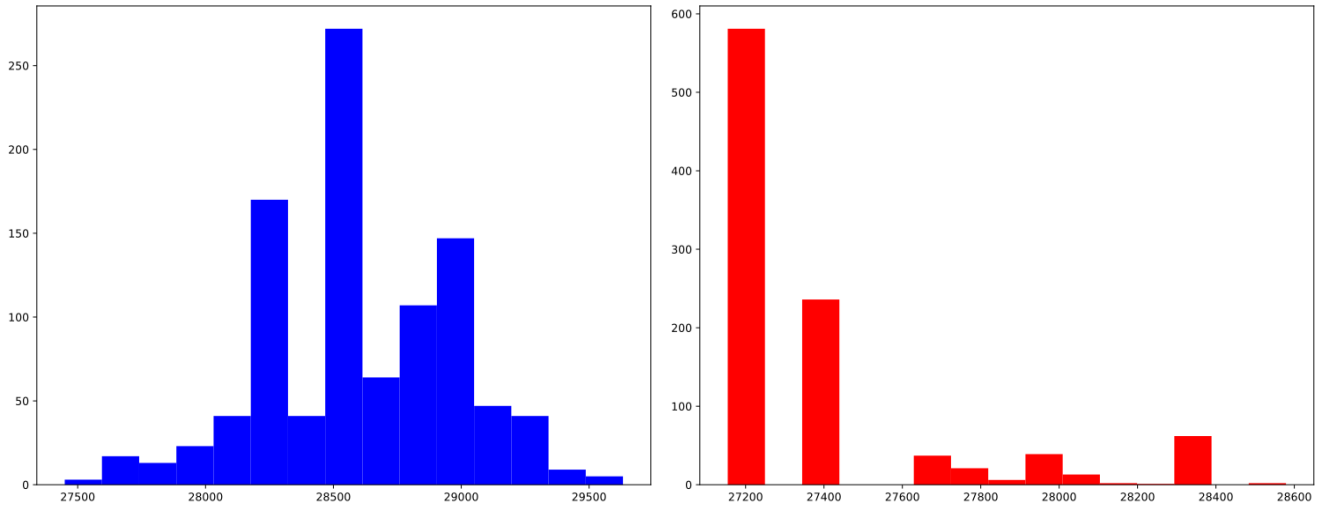
```
Final Results:

* Best Route Length after Genetic Algorithm --> 28276.34383

* Optimized Route Length after 2-Opt Optimization --> 27154.4884

* Optimized Route --> [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, 5, 0]

* Algorithmic Time Taken --> 0.045 minutes

* Total Time Taken --> 0.048 minutes
```

(b) Final Results

Figure 1: Tour 29: Plots and Final Results

For this specific tour length, as asked, it was run 1000 times and for the final results obtained, the histograms (Figure 2a and Figure 2b) were plotted. It can be observed that the variability in the results was low. The mean

and standard deviation for the mean fitness were: **28610.17463** and **368.67346** respectively. As for the best fitness, values of mean and standard deviation were: **27365.14786** and **335.84306** respectively.



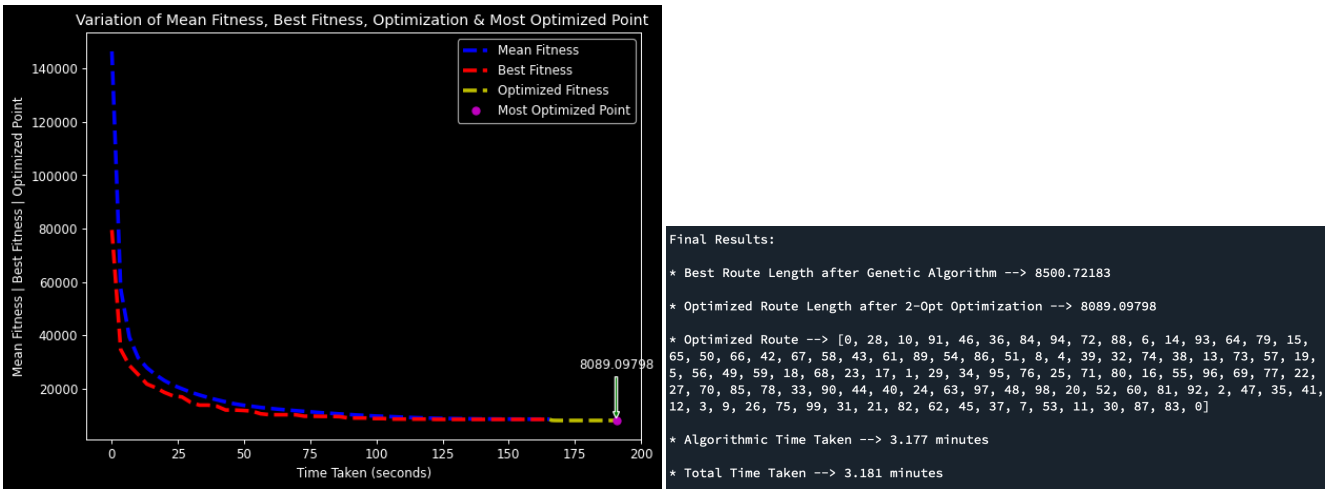
(a) Mean Fitness for 1000 iterations

(b) Best Fitness for 1000 iterations

Figure 2: Tour 29: Histograms for 1000 iterations

4.3 tour100.csv

From Figure 3a and Figure 3b, it can be seen that that the iterations ran smoothly and quite fast as well (within the time limit of 5 minutes).The tour length obtained (8089.09798) is better than the simple greedy heuristic value of 8636.5 and also not that greater than the optimal one with the difference being less than 800. For a time limit of 5 minutes, the solution seems pretty good. Just for information, the best tour length obtained from all the trial runs (within 5 minutes) was actually even better (somewhere around 7700) than the one obtained here. The difference is due to the variation in the initial population.



(a) Convergence: Mean Fitness, Best Fitness, 2-Opt LSO

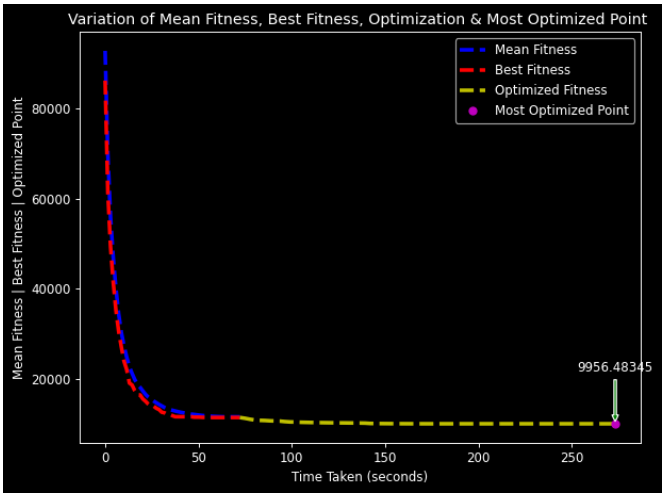
(b) Final Results

Figure 3: Tour 100: Plots and Final Results

4.4 tour194.csv

This is where the fun starts due to the large tour of 194 cities. Because of this, the calibration for this took a long time. Firstly this was due to the long computation time and secondly, it was not that easy to determine the interplay of several parameters and how they were influencing the results. Nonetheless, more time was spent analyzing this tour. The most important thing was to maintain the balance between exploration and exploitation in the case of a large tour. What this means is to strike a balance between the number of iterations of the genetic algorithm and the number of times 2-opt local search ran on the best solution. After much experimentation, the final values were decided upon. The optimal tour length is far better than greedy heuristic one and within a difference of 900 of the optimal value (9000). Figure 4a and Figure 4b below prove the point in case. It needs

to be mentioned that this is not the best tour length overall. During trial runs at my end (for a time limit of 5 minutes), the best solution obtained was around 9500, which is more closer to the actual optimal value of 9000. The reasoning behind the difference is related to the initial diversity of the population.



(a) Convergence: Mean Fitness, Best Fitness, 2-Opt LSO

```
Final Results:

* Best Route Length after Genetic Algorithm --> 11346.58168

* Optimized Route Length after 2-Opt Optimization --> 9956.48345

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

* Algorithmic Time Taken --> 4.555 minutes

* Total Time Taken --> 4.559 minutes
```

(b) Final Results

Figure 4: Tour 194: Plots and Final Results

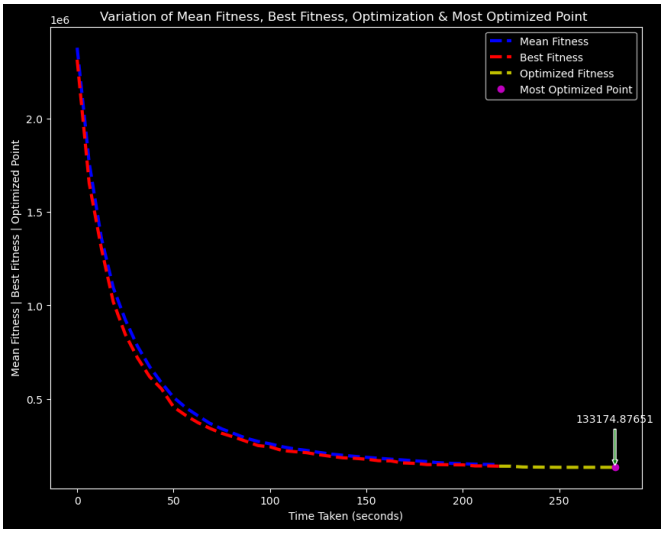
4.5 tour929.csv

This was naturally the most challenging tour to solve using the genetic algorithm due to the large number of cities (1000). The plots obtained are depicted Figure 5a and Figure 5b. As expected, to break the barrier of the greedy heuristic (113683.58) was in itself quite a challenge, let alone the most optimal one 95300). That said, the resulting tour length in my case was around 133174.87. Once again, I would like to point out that the best solution from all the trial experiments with varying parameter values did beat the simple greedy heuristic and was 113632.60. However this was without the time limit of 5 minutes. To reach this value, I had to again balance exploration (genetic algorithm) and exploitation (2-opt local search) and only after many trials, a good combination was deduced. But those parameter values couldn't be used for the final submitted code due to the 5 minutes constraint. For that, I, therefore, refined the values once more so that the code finished within 5 minutes. For the submission, the aim was to have a strong convergence rate (using sequential crossover and inverse mutation) and obtain a final solution (although full convergence was not attained as the code was constrained by the number of iterations). Accordingly, more emphasis was laid on exploration and less on exploitation (number of 2-opt local search were reduced compared to other tours). This sharp convergence is very well evident from the plots. The number of iterations too were reduced to around 35, again, due to the large size of the candidate. All in all, I believe that it was a challenge to even come within the range of the greedy heuristic. That was the set goal for this case where optimizing the two parts was everything. It was deemed more important to have a solution (albeit not properly converged and optimized) within 5 minutes which might be a bit off the mark (greedy heuristic) rather than not obtain anything and just have couple of iterations of the genetic algorithm due to extremely good but time-consuming (929 cities) computations. However, multiple trial runs were carried out and better solutions were obtained for sure for the case of without the 5 minutes criteria. The best result was not too far from the optimal one, with the difference between them being around 17000.

5 Critical reflection

This project equipped me with tremendous understanding of how these meta-heuristic genetic algorithms work. As everything was implemented on my own, starting from scratch, I learnt a lot. Some of my key takeaways from this project are as follows:

- Genetic algorithms are super helpful in obtaining fantastic insights about some of the very difficult problems (due to a large number of combinatorial variations) that I believe are otherwise not that easy to even approach and solve. The solution, though approximate, is still a good estimate of the actual solution and can be used for further practical application, decision making and analysis. Also, formulating a problem in terms of the genetic algorithm framework is not that convoluted. The problem can be set up quite easily. This point is evident from this project. In fact, it is also possible to use genetic algorithms as a good starting point to obtain a solution which can then be further optimized using other methods. Easy to understand, formulate and implement are some of the positive points associated with genetic and evolutionary algorithms. One of the best solution nature has provided us with!



(a) Convergence: Mean Fitness, Best Fitness, 2-Opt LSO

```
Final Results:

* Best Route Length after Genetic Algorithm --> 139149.97683

* Optimized Route Length after 2-Opt Optimization --> 133174.87651

* Algorithmic Time Taken --> 4.648 minutes

* Total Time Taken --> 4.651 minutes
```

(b) Final Results

Figure 5: Tour 929: Plots and Final Results

- Two major issues with genetic algorithms are - finding the most appropriate operator and parameter values and second, time involved for computation etc. Calibration of all the parameters to get a set of optimal values that allow a good quality of solution to be produced in a definite amount of time can be very challenging. This diversity is inherent to genetic algorithms and such large number of variations/possibilities (with parameters) proves to be a significant issue that needs to be tackled properly. And the other issue requires good computation resources for ensuring that the iterative procedure and optimization are carried out smoothly and quickly. The computation cost as well as resources (time) required needs to be taken into account. These two issues may not be easy to completely eliminate always. Depending on the requirements, the effect of these two factors can vary, however with a carefully planned approach, the problems with such issues can be reduced a little bit (for instance with multi-threading, computation load can be reduced).
- I did read about other existing approaches for solving the TSP problem. Many different algorithms have been developed to solve this problem. But as I have not done a proper study to compare all the approaches that exist for solving this TSP problem, it is not so easy to make a detailed analysis. Nonetheless, I do believe that using genetic algorithms for tackling combinatorial optimization problems is one of the good ways to get close to the actual solution. Decision regarding which approach is finally chosen is dependent on a lot of factors and there can never be *one shoe fits all* approach. Many considerations have to be accounted for and then based on the type and nature of the problem, existing resources, constraints, expertise etc., a much better decision can be taken. Using evolutionary algorithms surely can be one of the best possible candidates for such combinatorial problems.

In a nutshell, I now fully comprehend as to how genetic and evolutionary algorithms work and implemented. I believe I am in a position that I can use my gained knowledge, skills and most importantly practical experience that I acquired from this project, to work on another problem, for which evolutionary/genetic algorithm seems to be the best candidate, and work out a solution.

6 Other comments

To conclude, I would just mention that it was a very good opportunity to learn about Evolutionary and Genetic Algorithms. Practical learning is something that I always relish and the same happened in this case. Working on such a famous problem - TSP, it gave me a lot of confidence and good insight into how these meta-heuristic algorithms work. The project was a bit challenging but to be able to submit a working code with a good solution is a fantastic feeling. The course was super interesting involving hands-on-learning. For that and overall - Thank you Professor for everything!