Evolutionary Algorithms: Final report

Saptarshi Chakrabarti (r0775439)

April 26, 2024

1 Metadata

- **Group members during group phase:** Jacob Golub and Rasmus Jørgensen
- **Time spent on group phase:** 10 hours
- **Time spent on final code:** 60 hours
- **Time spent on final report:** 6 hours

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

The following changes have been made since the group phase of the project:

1. *Structure of population* : NumPy arrays instead of individual lists.
2. *Initialisation* : A mix of random and greedy permutations instead of fully random permutations.
3. *Selection*: k-tournament instead of exponential selection
4. Recombination operators : Added order crossover in addition to partially mapped crossover.
5. *Mutation operators* : In addition to inverse mutations, swap mutation has also been added.
6. *Diversity promotion* : Fitness sharing has been added since the group phase of the project.
7. *Convergence criteria* : changed to break the loop when mean fitness does not improve.

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

Some additional changes were made from the group project including stepping away from an object-oriented approach and taking a more functional approach. This decision was entirely based on my comfort regarding ease of debugging the code.

Additionally, during the first several attempts some crossover operators were generating invalid solutions. When trying to fix this problem, I realised that a greater focus is needed on the fitness of the initial population being as close to the heuristic solutions. This is especially the case for larger instances of the TSP and is reflected in the initialisation of the first population set – generating the population with an increasing mix of greedy solutions for higher problem instances. This measure helps for relatively early convergence to an acceptable solution given the time and hardware limitations for the project.

3.1 The three main features

1. initiate_population function - As stated before, the initialisation phase of the project is a key part of the code as it favours more greedy solutions for higher problem instances. This includes calculating the mix of the greedy and random solutions.
2. crossover and mutate functions: The functions which implement the crossover and mutation operators on the population array are written to introduce as much diversity with each iteration as possible. While the functions themselves are not significantly unique - their purpose is to make the main loop clutter-free and easier to tweak in case changes are needed. Theoretically, the actual crossover and mutation functions (for instance: partially_mapped_crossover, order_crossover, swap_mutation and inverse_mutation functions) are quite standard and can be replaced without breaking the code or negatively affecting the performance of the algorithm.
3. Considering the time and CPU budget for the project, fitness sharing is not implemented in every iteration. The periodic implementation is done using a parameter gen_skip – this allows for a less computationally expensive strategy.

3.2 The main loop

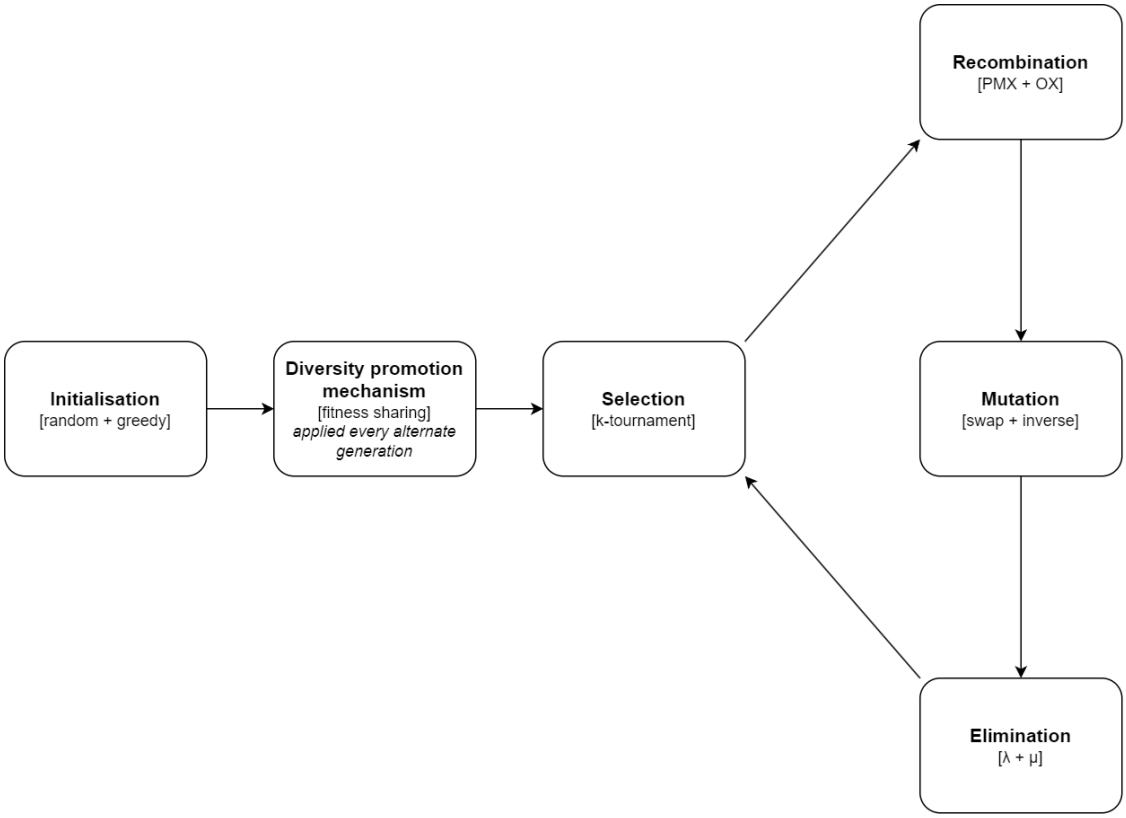

Figure 1: Design of the algorithm

3.3 Representation

The representation of the candidate solutions remains unchanged from the group phase of the project. The solutions are represented as paths where a path $2 \rightarrow 3 \rightarrow 1 \rightarrow 4$ is represented by the vector $(2, 3, 1, 4)$.

Other than this representation making it easier to implement the different operators chosen for the algorithm, this representation allows for minimal change from the group phase of the project and therefore more time to experiment with the other aspects of the algorithm. In addition to this, adjacency representation was considered but that would require re-implementing several aspects of the project which would not always guarantee faster or optimal performance.

Each candidate solution is constructed as a 1-D NumPy array, as standard operations on these are efficient. The entire population is built as another NumPy array of these solutions. The choice of NumPy array was motivated primarily by the ease of operation casting over the array and its compatibility with Numba library for JIT compilation advantages.

3.4 Initialization

As mentioned initially, the population initialisation is based on several factors. Most important factor of all was the time budget along with the availability of only two physical CPU cores. This introduces the requirement to converge on acceptable candidate solutions relatively fast, and, especially with fewer iterations in the case of large problem instances which contribute to expensive computation. A requirement I felt necessary for this was to initiate a population where the best candidate's fitness was as close to the heuristic solution as possible, so that lesser time and resources are spent exploring invalid paths generated randomly.

In the context of this algorithm, I have defined three population generation functions with varying degree of heuristic nature:

1. 'random_population': this generates random paths from the distance matrix as expected from the name of the function
2. 'greedy_population': this chooses a starting point at random and then generates a path using a nearest neighbour approach. However, for the sake of diversity, I have tried to implement this in a way that multiple paths do not share the same starting point.

3. 'greedier_population': this takes the heuristic nature a step further for extreme cases as it selects the starting point based on the shortest path from the distance matrix.

The final version of the initial population is then put together using the 'initiate_population' function. This is done by generating a combination of individuals with varying degrees of random to heuristic path ratios. This ratio is determined by the size of the problem instance. For example, tour50 is initialised with a 1:1 random to greedy population ratio. Whereas, for higher problem instances like tour1000, an entirely greedy population generated using 'greedier_population' function.

The ratio of the mix of random to greedy solutions in the population is calculated by scaling size of problem instance to fraction between 0.75 to 1.0, with the exception of very small problem instances where the 1:1 ratio is maintained because there is no sufficient gain from a strongly heuristic set of starting solutions.

The exploitative behaviour introduced by such heuristic population generations are offset by more aggressively introducing mutations to a large percentage of the population as well as using fitness sharing at regular intervals to promote diversity in the population. I took the decision to not implement a local search operator partly due to the need for maintaining diversity among other reasons (which are discussed in later sections).

The decision about number of individuals in the population is aimed at striking a balance between a population size small enough to allow for cheap computation in the case of large problem instances and being large enough to capture a good variety of solution candidates to begin with. To ensure that population enrichment schemes do not take over the entire population, the mutate and elimination operators are designed to only return individuals that are not present in the original population. The choice of every operator in the algorithm is primarily motivated by the need to maintain as much diversity as possible in the population.

Initially, I tried to form a mix of random, greedy and greedier initiation for all initialisation of population instances. However, it was generating some very strong candidates which limited the solution search to very localised spaces. So, this method was discarded in the final version.

3.5 Selection operators

For the selection operator, I deviated from the group phase and decided to go for k-tournament selection. This allowed for some degree of randomness in the selection process. The choice for this operator was because discrete values of k lend better and easier test-ability to the algorithm.

My implementation of k-tournament requires two parameters to be specified:

- k-value for the number of rounds of the tournament
- number of parents that need to be selected from the population

While no advanced methods were used to vary these parameters across iterations, the values used for these parameters were decided after several runs and opting for a value that helps obtain acceptable solutions for all the problem instances provided. I decided to go for k-tournament over exponential selection in the final version of the project entirely because of the simplicity of the operator.

3.6 Mutation operators

Inverse and swap mutations were implemented in this algorithm. The choice of these operators was based on their simplicity and tendency to only change a fragment of the individual instead of destroying the entire sequence. This introduces randomness but not too much randomness that would render the application of the operator pointless.

While such operators can possibly be controlled with parameters like subtour length and mutation probability - the only one used here was mutation probability. An initial effort was made to vary this probability value every couple of iterations, but there was no meaningful gain from this which is why I decided to discard it in the final version.

Scramble mutation was not considered for this as they introduce too much randomness. Such an aggressive operator is a bad choice for the travelling salesman problem.

3.7 Recombination operators

Both partially mapped crossover (PMX) and order crossover (OX) are aimed at preserving orders and relative orders of their parent genes respectively while still introducing diversity. Both these operators can be controlled by parameters like sub-tour length and crossover probability. The sub-tour length decides the length of the fragment that will be mapped or selected for identifying the order in. In my implementation, I have only used crossover probability and the sub-tour lengths are decided by random choice. In some initial implementations, I

had the crossover probability change over iterations but later decided to discard this idea as I did not notice any significant gain from this.

PMX and OX ensures that the relative orders of elements within the selected subset is maintained, contributing to the preservation of good building blocks from both parents. Even if both parents have very little overlap, PMX can still create diverse offspring by combining these shared genes with the corresponding elements from the other parent whereas OX can create offspring by preserving the order within this subset.

In a good number of attempts, I tried edge recombination and two-point crossover operators, however my implementations were relatively inefficient for larger problem instances. This led me to discard them in the final version of my code.

3.8 Elimination operators

The $\lambda+\mu$ operator (with some modification) has been retained since the group phase of the project for elimination. Instead of combining the selected parents and offsprings, my implementation combines the entire population before the selection step with the entire set of population after all variations (crossovers and mutations) are made. This decision has been made to avoid early convergence in the case where not enough unique individuals are generated through the variations and to preserve good candidates from the previous generation even if they were not necessarily selected for the crossovers or mutations. This allows the output of this function to be directly passed on as the population for the next iteration of the evolutionary loop without needing further modifications outside of the function.

Other than this, no other modifications were made, or different operators were attempted as this strategy offers an efficient balance between exploring diverse regions of the search space and exploiting promising solutions.

3.9 Local search operators

In the penultimate version of my code, I included a k-opt operator. However, I did not find my implementation efficient enough to make a meaningful change to the solutions. Considering the computationally expensive nature of this operator and the efficiency of the rest of my algorithm in combination of the constraints set for the project - I decided to discard this in my final version. The solutions to the problem being in a discrete domain - I did not consider other local search operators like simulated annealing as they are better suited for continuous domains.

3.10 Diversity promotion mechanisms

I implemented fitness sharing as a diversity promotion mechanism in this algorithm. The function is only implemented every alternate iteration. This works by adjusting the fitness values of individuals based on their similarity. It penalises individuals with neighbours in the population, encouraging a balance of representation and a sure way of introducing diversity.

As my initialisation methods heavily lean on exploitation and the elimination operator already promotes only the fittest candidates to the next generation – this operator helps address premature convergence. A distance parameter (sigma - σ) and a shape parameter (alpha - α) need to be determined for this mechanism. However, due to lack of time I have implemented the best case from an arbitrary guess of optimal values. These values are updated every iteration as follows:

- σ is updated based on the 1/1000 * best fitness value of the population
- α is originally set at a static value which is then decayed with every generation by 0.99

3.11 Stopping criterion

While I have set an upper limit on the maximum number of iterations, they are never reached given the time budget of the project. However, an additional criterion I have added (which is also hardly ever met within the time frame) is if the mean fitness of the solutions remains unchanged over 50 generations.

3.12 Parameter selection

There are only few parameters that are set to be automatically determined.

- The ratio of heuristic to random solutions in the population is based on the size of the problem instance which is scaled down to a range between 0.75 and 1.0 for tour lengths above 100.
- The distance parameter (σ) for fitness sharing is determined by the Euclidean norm of the best fitness value of the population divided by 1000.

- The initial population size, $lambda$ (number of parents to select) and k value for k-tournament have been selected by manually evaluating combinations of these parameters which balances exploration and exploitation for all sizes of problem instances with relatively decent speed. My preference was to keep the algorithm as simple as possible and decided against self-adaptivity which would necessitate further testing. This also meant that I have not used parameters which work well for some problem instances but not so much for others.
- Crossover and mutation probabilities are intentionally kept high to aggressively introduce variations in the population - which is especially useful for larger problem instances where the first set of solutions are compiled with a heavy mix of greedy solutions.

3.13 Other considerations

In each iteration, I have chosen to apply two kinds of crossovers and mutations as aggressively as possible to counteract the heuristic nature of the population that is passed on as the initial population at the beginning of the main loop. These implementations are specifically realised through the crossover and mutation functions which are named explicitly the same.

For the purpose of simplicity (during my trial runs), the key convergence test is placed at the bottom of the main loop instead of being assigned to the variable dedicated to store the same.

4 Numerical experiments (target: 1.5 pages)

4.1 Metadata

Parameters - fixed for all problem instances:

- (maximum) iterations = 2000
- population size = 120
- lamda (parents to be selected) = 40
- gen_skip (number of generations to skip) = 2
- crossover probability = 0.95
- mutation probability = 0.95
- mutation population percentage ((bottom) percentage of population to mutate) = 0.90
- random seed = 42
- k (for k-tournament) = 4
- alpha = 2.5 (this is decayed over every successive iteration)

Parameters - determined by problem instances:

- greedy mix percentage calculated from size of distance matrix:
  - tour50 : 0.5
  - tour100 : 0.763
  - tour500 : 0.868
  - tour1000 : 1.0
- sigma - automatically filled from population fitness matrix

Specifications of computer system:

- CPU: AMD Ryzen 7 4800H - 8 cores at 2.90 GHz base clock speed
- Main memory: 16 GB
- Python version: 3.11.5

4.2 tour50.csv

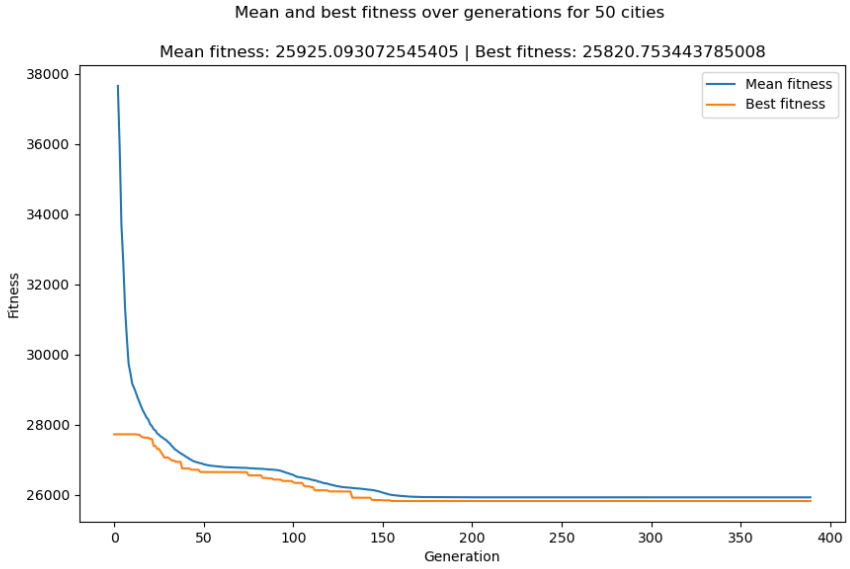

Figure 2: Convergence graph for 50 cities

The best tour length for 50 cities came out to be 25820.753 with the sequence of cities as follows:

$[25 \rightarrow 44 \rightarrow 12 \rightarrow 23 \rightarrow 28 \rightarrow 27 \rightarrow 36 \rightarrow 33 \rightarrow 34 \rightarrow 40 \rightarrow 17 \rightarrow 21 \rightarrow 26 \rightarrow 0 \rightarrow 32 \rightarrow 37 \rightarrow 29 \rightarrow 20 \rightarrow 49 \rightarrow 22 \rightarrow 6 \rightarrow 9 \rightarrow 43 \rightarrow 30 \rightarrow 48 \rightarrow 5 \rightarrow 8 \rightarrow 46 \rightarrow 35 \rightarrow 16 \rightarrow 4 \rightarrow 15 \rightarrow 47 \rightarrow 42 \rightarrow 11 \rightarrow 3 \rightarrow 19 \rightarrow 38 \rightarrow 10 \rightarrow 13 \rightarrow 7 \rightarrow 41 \rightarrow 14 \rightarrow 1 \rightarrow 18 \rightarrow 24 \rightarrow 45 \rightarrow 2 \rightarrow 31 \rightarrow 39]$

For the case of small problem instance like 50 cities, the algorithm converges relatively early given the speed of each iteration and total possible iterations in 5 minutes – around the 340^{th} iteration. Given that the population size allows for a large enough number of solution candidates, diversity of population is preserved over the iterations till the mean fitness does not change over 50 iterations. This is best seen through the difference in mean fitness and best fitness towards the beginning of the algorithm. The best solution seen over several runs of the algorithm indicate that the one shown here is not the global optima as better solutions have been obtained using other combination of parameters. However, this solution is obtained using parameters which can perform better than the heuristic figures provided before for all the benchmark problem instances provided. The obtained solution is 6.86% better than the heuristic solution provided before for all the benchmark problems.

Given the smaller shape of the population array $(120, 50)$ and circular references used in updating the population array, memory usage has been minimal.

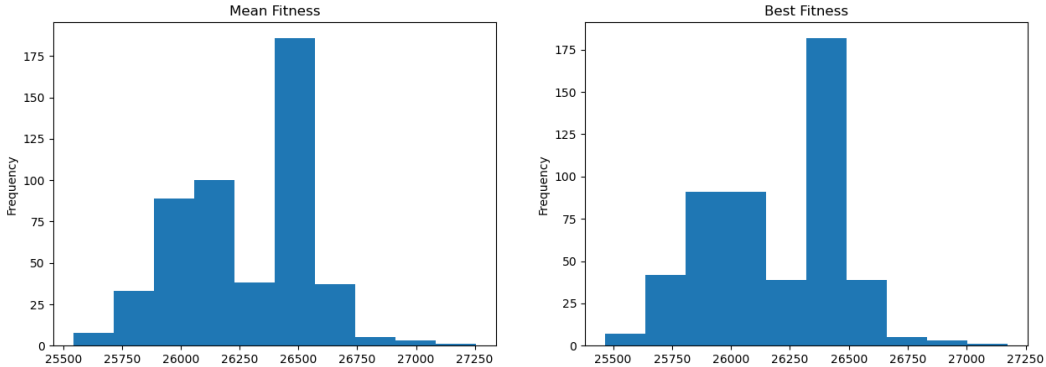

Figure 3: Histogram : final results from 500 runs

The data from running the algorithm on tour50 500 times suggests that the algorithm performs consistently on average, with a mean fitness that is relatively stable and a small standard deviation. The mean of the best fitnesses shows a consistency in identifying the best solution candidates. There appears to be a steady convergence towards optimal solutions, as indicated by the low standard deviation of best fitnesses. The variances indicate a

moderate degree of variability, with a greater range in mean fitness and optimal solutions, for both the mean and best fitness values.

The statistics of the final means and best fitnesses from 500 runs are as follows:

- Mean of mean fitnesses: 26272.0196
- Standard deviation of mean fitnesses: 284.1611
- Variance of mean fitnesses: 80747.5562
- Mean of best fitnesses: 26183.7032
- Standard deviation of best fitnesses: 294.6099
- Variance of best fitnesses: 86795.0384

4.3 tour100.csv

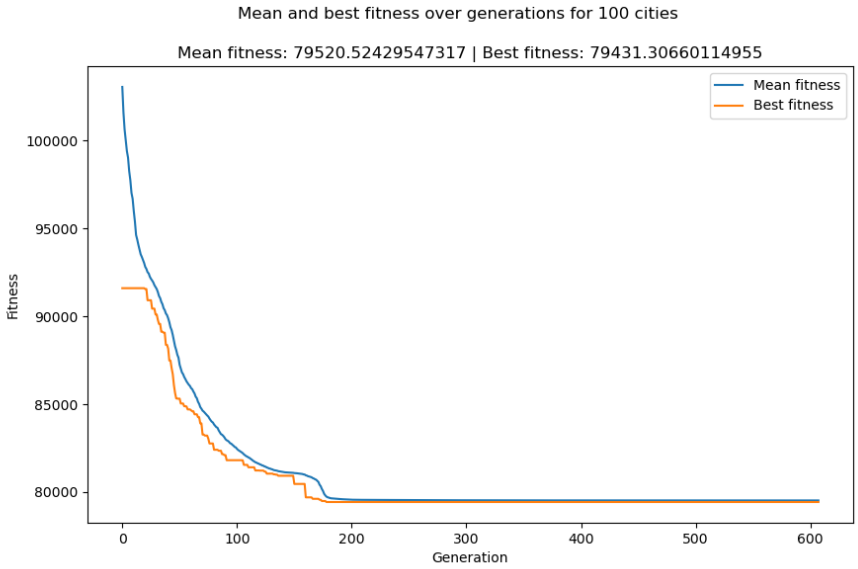

Figure 4: Convergence graph for 100 cities

The best tour length for 100 cities obtained is 79431.307

For this problem instance as well, the algorithm again converges relatively early given the speed of each iteration and total possible iterations in 5 minutes – around the 558^{th} iteration. Given that the population size allows for a large enough number of solution candidates, diversity of population is preserved before it almost converges around the 200^{th} iteration. The best solution seen here is one of the better solutions that I have found using all the combinations of parameters I have tried. This does not imply that better solutions do not exist or that this is the global optima for this problem. This can probably be corrected through better exploration strategies.

Given the smaller shape of the population array $(120, 100)$ - memory usage has again been minimal.

4.4 tour500.csv

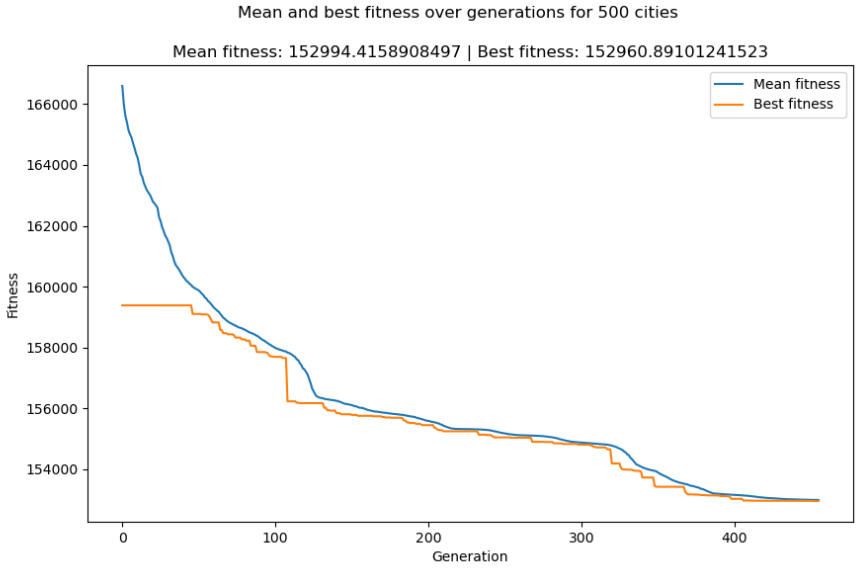

Figure 5: Convergence graph for 500 cities

The best tour length for 500 cities obtained is 152960.891

From the appearance of the graph the solutions seem to be converging before the loop hits the 5 minutes mark at the 446^{th} iteration. Given that the population size does not allow for a large enough number of solution candidates, the algorithm struggles to maintain diversity of population over the generations. The best solution seen here is still one of the better solutions that I have obtained during my other attempts. This also leaves space for better solutions that can be found through better exploratory strategies. Better solutions can probably also be obtained with longer run-times.

While the relatively larger shape of the population array $(120, 500)$ accounts for slower speed per loop due to the loop operations within the main evolutionary loop, it does not consume significantly large memory.

4.5 tour1000.csv

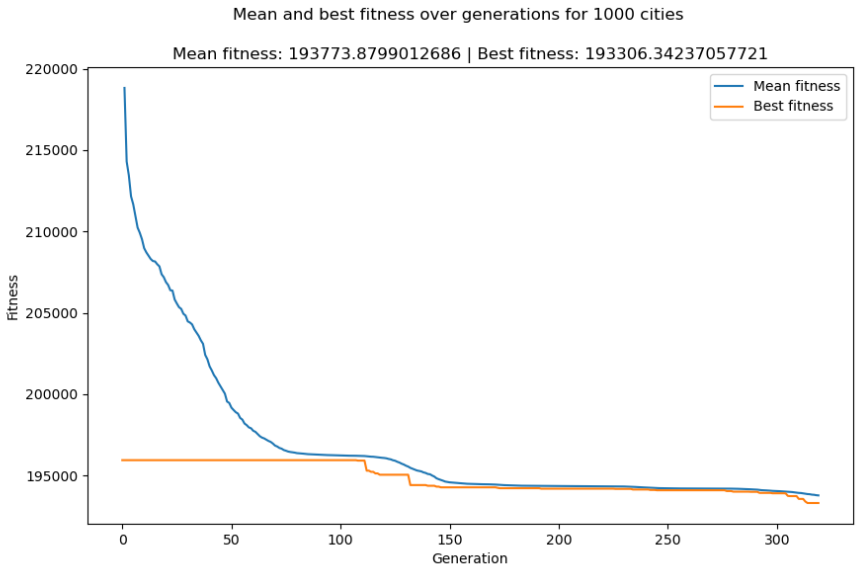

Figure 6: Convergence graph for 1000 cities

The best tour length for 500 cities obtained is 195107.506

The convergence graph indicates the solutions slightly diverge just before the loop hits the 5 minutes mark at the 329^{th} iteration. The smaller population size of 120 does not allow for a large enough number of solution candidates for the algorithm to be meaningfully exploratory. This combined with the fact that for this problem instance, the initialisation has been done through an excessively greedy approach - the algorithm struggling to maintain diversity of population over the generations comes as no surprise. The best solution seen here is still one of the worse solutions that I have encountered during my previous attempts. However, this does perform better than the heuristic solution provided previously. A much higher population size that accounts for more exploratory initial solutions combined with more run-time might yield better results. The choice of a heavily exploitative first population was made entirely to cut down some of the time used to explore the solution space enough to reach a solution which is at least as fit as the heuristic solution fitness.

The relatively larger shape of the population array (120, 1000) accounts for even slower speed per loop, it consumes a moderate amount of memory for the intermediate calculations. However this memory consumption is still not significant for most modern computers.

5 Critical reflection (target: 0.75 pages)

Three main strengths of this evolutionary algorithm:

1. Initialisation favouring greedy solutions in case of large problem instances help cut down on runtime overheads which would have otherwise been spent arriving at populations with fitness levels similar to the initial population.
2. Introducing variations to the population aggressively lend more exploratory features to the algorithm which allows to further offset the heuristic nature of the initialised population.
3. Population size being set 120 strikes the right balance to capture diverse enough solution candidates which combined with heuristic initialisation typically has shown to include quite good solution candidates in the first generation itself while being a low enough number to not get slowed down for large problem instances where each solution candidate is quite large to work with.

Three main weaknesses of this evolutionary algorithm:

1. The lack of self-adaptive features most likely result in sub-optimal solutions as sub-optimal configuration of parameters are selected.
2. A good part of the recombination and mutation operators are left to random chance and there is less control on how it is applied - which may be in a questionable area of optimality. This is most evident in larger problem instances where the algorithm struggles to maintain population diversity a few iterations after initialisation.
3. The lack of an efficient local search operator is visible through some of the more disappointing runs of the code which produced worse results than the ones reported here.

The problem studied in this project being an asymmetric travelling salesman problem is uniquely suited for evolutionary algorithms as combinatorial problems in discrete solution spaces. The population-based approach (as is the case with genetic algorithms) allows for simultaneous exploration of diverse solution candidates (in this case - paths), enabling the algorithm to discover optimal solutions through relatively simple methods.

One of the things that I liked about the modular nature of an evolutionary algorithm is that it lends a great level of both sophistication and simplicity to the approach. For instance, while parameters like selecting sub-tour lengths for crossover and mutation operators allow for great control over the variations introduced in each iteration of the evolutionary loop, it is also quite easy to simply switch out an operator with a different one for the kind of performance one is looking for. This has been especially surprising for me given how challenging I found maintaining diversity in the population to be. Perhaps the design of the building blocks of the algorithm needs more attention than just the implementation of it - something I missed out during my very chaotic approach to this project.

As someone who has seldom written code keeping time and computational resource limitations in mind, my approach to "how do I make this population converge to an acceptable solution within the time and resource budget allocated" has taught me a new way of thinking about algorithm design and problem solving.