Evolutionary Algorithms: Final report

Chidi Nweke (r0786701)

December 31, 2021

1 Metadata

- **Group members during group phase:** Sai Niranjan and Mathijs Dom
- **Time spent on group phase:** 10 hours
- **Time spent on final code:** 40+ hours
- **Time spent on final report:** 7 hours

2 Peer review reports

2.1 The weak points

1. Time budget: in the group phase we used both a very small population and had strong fitness based selection. This caused the algorithm to converge very quickly.
2. Purely random initialisation: They warned us that as from tour 250 there will be a lot of unreachable cities and told us to consider strategies to counteract this during initialisation.
3. Fitness based selection: They had more criticism on the implementation of the scheme, code wise, than the idea itself.
4. Parameter K too high: very related to point 1. Our algorithm was converging too fast.
5. Weak mutation operator: we simply swapped to cities. This doesn't lead to a lot of exploration as they rightly pointed out.
6. Small population size: this is again related to point one, with a larger population we could see more of the search space and use the time budget better.

2.2 The solutions

1. **I fixed the population size at twice the amount of cities**. This amount allows me to find reasonable results for the smaller problem sets while scaling decently to the bigger problems. Especially at tour 1000 this is the case.
2. **The population is initialised with a greedy strategy** that sets each city to position 0 and then takes the shortest path. This allows for a decent starting population. A local search operator is also used at this step.
3. The parameter K, as were other hyper parameters were continuously tweaked to find better results. A higher K was used in the end to prevent extremely fast convergence.
4. I ended up implementing five or six different mutation operators. Out of all of them inversion mutation worked the best.

You could argue that the small population size suggestion wasn't sufficiently addressed for the smaller problem sizes. I did experiment with a formula such as taking the maximum of 500 and twice the total amount of cities but a larger population didn't really bring better results on the smaller problem sets.

2.3 The best suggestion

All suggestions had their merits, hence why I actively tried to incorporate what they mentioned. I referred to them multiple times when I got stuck trying to solve the bigger problem sets and I would definitely have struggled more without them.

One of the suggestions that I liked the most was one we gave to both teams: **using an individual class**. Using this approach you can easily store all individual objects into the most relevant data structure that fits your implementation. This approach also makes sure you don't have to recompute fitness values several times and makes using self-adaptivity schemes a lot easier as well. This is the scheme that we used during our group phase but I decided to do a drastic change from this.

3 Changes since the group phase

1. Biggest change: **Class based approach to a pure Numpy based approach**. This change has less to do with evolutionary algorithms, but more with efficiency. A key idea throughout this project for me was to create a framework that is fast and scalable from a computation point of view. For this reason I chose to use **Numba, a JIT-compiler for Python**. It comes with the advantage that it is several orders of magnitude faster than my previous solution and allows for **multithreading** (more on this later) but comes with the disadvantage that it is **very inflexible** as to what kind of code can be compiled. For this reason I had to **rewrite the entire algorithm after the group phase** and also move away from the object-oriented approach I was using. This is key because on the very large problem sets being able to get a reasonable amount of iterations in 5 minutes is one of the biggest concerns.
2. Purely fitness based selection to K-tournament elimination. Both schemes favour high fitness but the latter at least has some level of randomness in it.
3. Different mutation and recombination operators: I tried to implement most of both that I found in the book, research papers and the lectures. I ended up sticking with inversion mutation and ordered crossover (OX) instead of swap mutation and PMX. OX makes a lot of sense over PMX as it tries to pass on cities with respect to their relative position to each other while PMX passes on information based on their absolute position. It matters more that two cities are (not) next to each other than whether or not they are in the front or the back of the tour, which would matter more for a problem like knapsack.
4. All parameters were tuned and changed.
5. **Local search operators**: both a full k-opt and a less expensive variation.
6. **Elitism**
7. Initialisation strategy

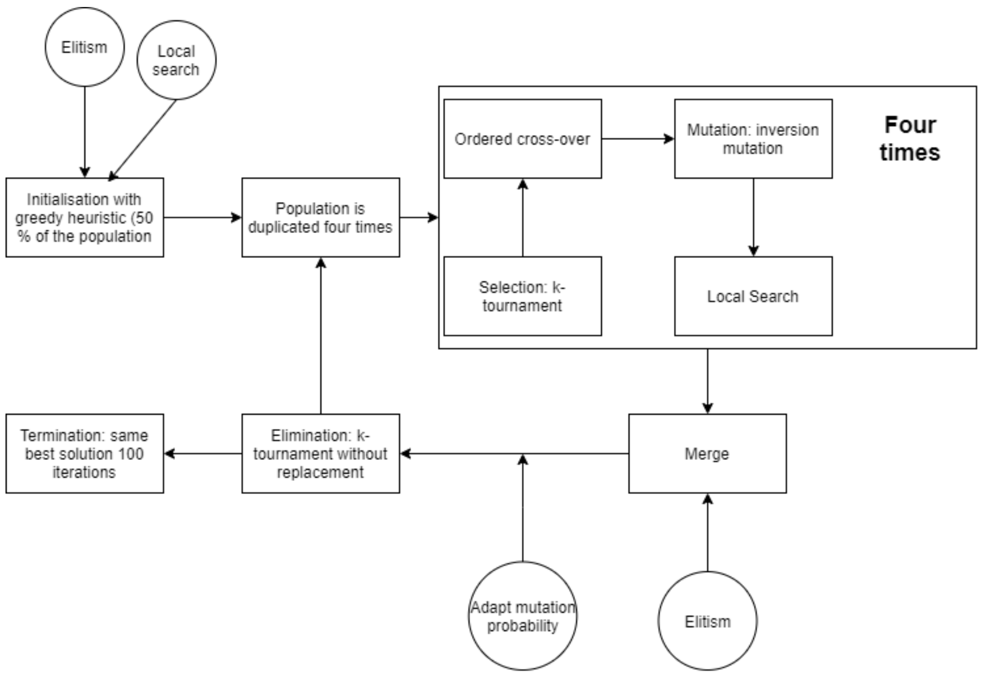
Essentially due to the first point there was **a near complete overhaul coming from the group phase** to the individual phase.

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

4.1 The three main features

1. Local search operators
2. Initialisation strategy
3. Multithreaded approach: increased chance for the mutation of parents to offset elitism.

4.2 The main loop


Elitism
Local search
Initialisation with greedy heuristic (50 % of the population
Population is duplicated four times
Ordered cross-over
Mutation: inversion mutation
Four times
Selection: k-tournament
Local Search
Merge
Termination: same best solution 100 iterations
Elimination: k-tournament without replacement
Adapt mutation probability
Elitism

4.3 Preliminary remarks

For some problem sets I tested heavily on, such as tour29 and tour250 I found more or less optimal parameters for my algorithm. Using them on tour29 I nearly always find the global optimum or land around it. For tour 250 my results with these parameters hover around 430000 but have reached 42000 as well. The parameters that have the most influence on this algorithm are: **maxDepth, K (elimination) and the mutation probability**. Keeping all other parameters equal, tour29 and tour250 favoured a different starkly different mutation probabilities.

The report will not reference nor use these exact parameters values as they are too situational, I will do my best to lay-out a framework that produces reasonable results on *all* problem sets even though more specialised ones would outperform these as this setting mirrors a real-world setting more.

4.4 Representation

I chose to represent the population as a Numpy array. Aside from this array I also keep track of an array with the individual's corresponding fitness values. Ideally I would have represented the the population as a list or 1-D array of objects but that would have made the specific approach I took impossible.

4.5 Initialization

My initialisation strategy starts by putting each city at position zero, after this the shortest path from that city to the next and onwards are selected. Considering my total population size is just twice that of the total amount of cities this means that **the greedy initialisation method takes up half the initial population.** This is not ideal, this makes it so that the population doesn't have a lot of diversity in the beginning but was done intentionally. I set the population size by looking at the smallest size the easier problem sets could still achieve a reasonable result. The reason is that during my experiments I found that **bigger population sizes did not result** in better results and they made tour 750 and tour 1000 intractable. A big part of my algorithm, this will be expanded on later, is the fact that parents can be mutated four times. This partially offsets the negatives of this approach.

An initialisation strategy like this is also part of a trade-off, computing a greedy heuristic for a TSP problem virtually requires no knowledge of combinatorial optimisation or meta-heuristics while being very efficient. To run, this means that any complex algorithm should always outperform this benchmark, otherwise it is not worth running. Putting all of them into the population ensures you will always outperform the heuristic but has a lot less diversity. This is quite similar to the bias-variance trade-off.

Afterwards I do a local search operator, rand-opt, which I will describe later on.

After initialisation I also keep track of a number of the best individuals. I pick 3 or 1% of the total population, whichever is larger.

4.6 Multithreaded approach: 'copying' parents

Before selection happens I take the total population and do recombination and mutation four times (as we were **restricted to 2 physical cores with 2 threads per core**). Each thread handles an entire 'copy' of the population. This has a multitude of advantages:

1. It is faster: I tested it with and without multithreading and the former outperformed the latter. It especially outperformed doing the same with multiprocessing, which was slower than using 1 core because of the overhead.
2. Crucially: **Parents can be mutated several times**. Each thread receives a copy of the population. Selection and recombination follow after this, the children and parents are merged. Afterwards mutation with a given probability is applied to the entire set. This happens in each thread hence why parents can be mutated several times and more randomness is added to the algorithm.
3. not in the scope of this project but this could easily be extended to more threads and different schemes could be employed per thread. I attempted the latter but it resulted in less desirable results and was harder to implement.

The main disadvantage of this approach is simply implementing it. Python does not support multithreading out of the box due to the global interpreter lock. This is why Numba was a key part of this approach: it allows JIT-compiled Python to run on different threads.

4.7 Selection operators

I started off with k-tournament selection. Before trying out any other operator I used varying levels of k, going from 1 to very large numbers. **None of these impacted the performance of the algorithm significantly**. Randomly sampling parents from the population did not necessarily result in a worse algorithm. This is most likely due to the fact that this approach doesn't have a lot of diversity to begin with. Because of this reason I purposely chose not to implement any different selection operator.

If this wasn't the case I would have used a scheme such as sigma scaling instead. This was one of my first ideas going into the individual phase but due to the things listed above I chose not to use it.

4.8 Mutation operators

Inversion mutation is the main driver of my approach. It introduces the most randomness into the population. Setting it slightly too high potentially destroys all interesting formed children from recombination. Setting it too low ensures the algorithm will have no diversity and get stuck in a local minimum. Furthermore all problems required a different mutation rate.

As I previously mentioned, I was able to tune the mutation probabilities *in hindsight* with trial and error which is not the right way to work in my opinion. The way I set the mutation rate is as follows: it starts at 0.5, each iteration where the best solution is the same as in the previous iteration it increases by 0.01 to a maximum of 0.8. If the best solution is different than that of the previous run it decreases by 0.02 to a minimum of 0.3. Essentially, **if the algorithm is converging it tries to force it way out of a local minimum by increasing the mutation rate**.

This solution performed worse than problem specific hyper parameters but outperformed using the same mutation rate on all problems hence why it was chosen.

4.9 Recombination operators

I simply use OX as my recombination operator. I implemented most of them that I could find in survey papers, the book and the slides (PMX, CX, ...) but OX performed the best averaged out over a lot of times. This makes intuitive sense as I already explained, it tries to keep neighbouring cities intact instead of preserving their absolute position like PMX would do. In certain experiments I ran I used a mix of operators: one offspring would be created with OX and the following with PMX, this also resulted in suboptimal results.

I do not use any self-adaptivity here either. My recombination operator simply takes two parents and produces two offspring. **The main line I explored during this was using a multi-offspring approach** inspired by a paper I read. Selection can be an expensive operation and producing 4, 12 or even more offspring per parent seemed like an interesting idea. In one experiment I used OX to create two children and then used OX to create two more from them. I also tried using OX followed by PMX and vice versa. **All of these produced worse results than just using OX.**

I did not consider any approach with an arity large than 2.

4.10 Elimination operators

My elimination operator is a basic k-tournament elimination without replacement. I tried doing the same approach as with the mutation probabilities, if the best solution did not change after a set amount of iterations I would gradually start raising k. As soon as it dropped in one variant I would reset k and in another one I would drop it by 2 or 3. Again, all of these variants performed worse than setting it in advance to 6, 7 or 8.

4.11 Local search operators

local search operator heavily inspired by k-opt I call rand-opt in the code. Each individual samples a $maxDepth$ cities, afterwards for each of these cities $maxDepth$ cities are sampled, swaps are carried out between these. At each step the fitness is computed and the best one is returned. Essentially, it **samples from the neighbourhood without fully constructing it.** This is done for two reasons: a full k-opt was very slow on large problem sets and it caused the algorithm to converge too quickly on the small problem sets.

It's main parameter $maxDepth$ has both a drastic impact on speed and performance. Setting it too high makes the algorithm converge faster and slows performance down considerably. Setting it too low makes the results worse. It's default setting is 20 and that works well for all problem sets. In a different scheme I had it vary in the same way that the mutation probability and K did. **In hindsight getting all three aligned when varying between iterations would be the best solution but is another optimisation problem in its own right.**

4.12 Diversity promotion mechanisms

I tried two diversity promotion mechanisms: converting the tours into a string representation like 53214, storing this in a set and using $O(1)$ operations to check if it is already in the population or not. A second more advanced one was converting them into a string, sampling from the individuals already in the population and computing the edit distance between them. After this I could use fitness-sharing.

Both approaches slowed down my code a thousand-fold on tour29 and are simply intractable for large problems so I stopped removed it from the algorithm.

4.13 Stopping criterion

The algorithm has 3 stopping criteria

- After 5 minutes.
- After the best solution has not changed in 100 iterations.
- After 1000 iterations.

4.14 Parameter selection

- Population size: $2xnumberofcities$. The smallest population size the easy problems could operate on. Allows the larger problems to get a decent amount of iterations in 5 minutes.
- Number of elites: 3 or 0.01 of the total population. Reinserting too many caused the population to converge too quickly. **Due to my extremely high mutation rate elitism is needed for the population not to spiral out of control.**

4.15 Other considerations

As mentioned previously, the parallel implementation is a key part of the algorithm and so is elitism.

5 Numerical experiments

5.1 Metadata

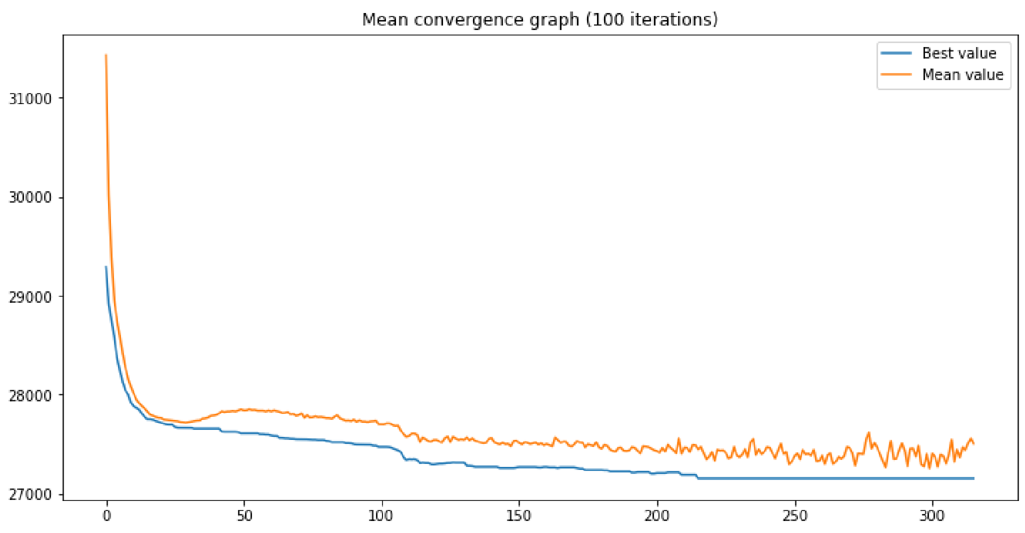
The numerical experiments were carried out with these parameters and not the problem specific ones.

The following parameters were used for all experiments

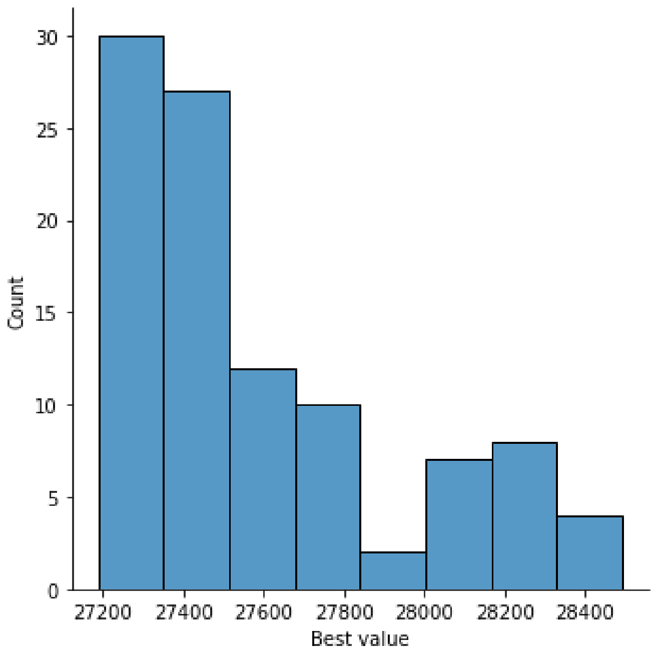
- population size = $2 * AmountOfCities$
- $K = 8$
- $NOffSpring = 1/4populationSize$ per core so $PopulationSize$
- $NumberOfElites = 0.01 * populationSize$
- $StartingProbability = 0.5$

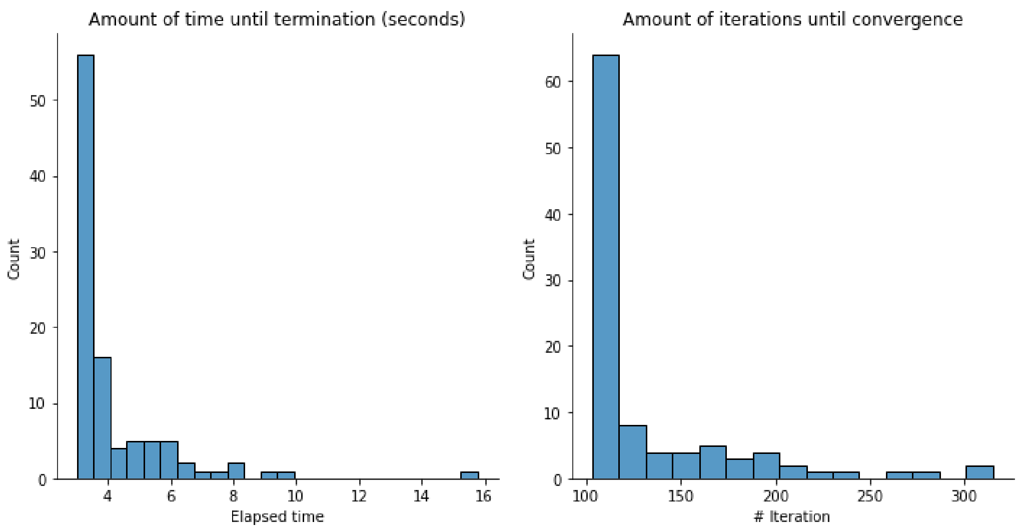
These experiments were ran on Intel® Core™ i7-8550U (4 cores total) Processor which has a frequency of 1.80 GHz per core. Since we were limited to 2 cores I only used 4 out of the 8 total threads. The system also contains 4 GB of ram.

5.2 tour29.csv



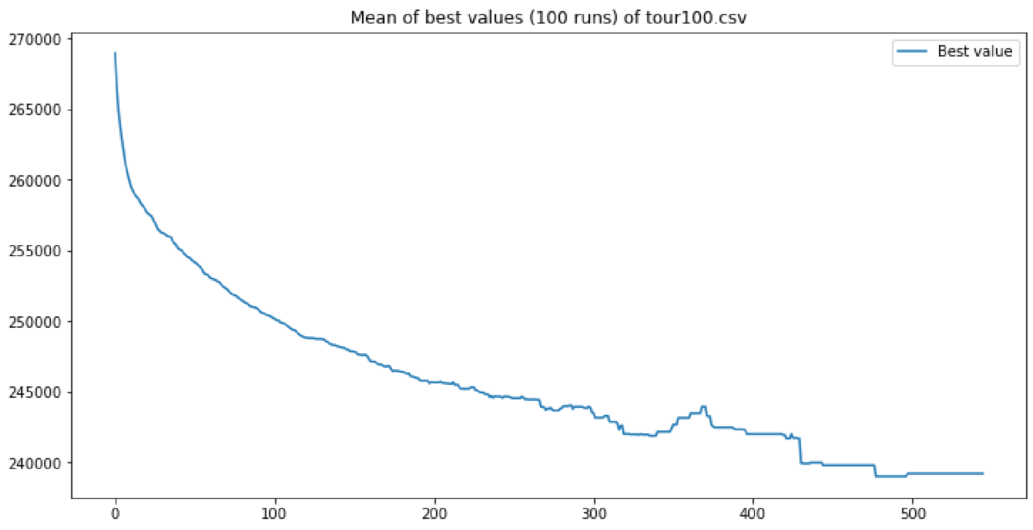
This plot shows the mean convergence graph over 100 iterations. It is clear that the algorithm nearly always finds the global minimum after a reasonable amount of iterations.

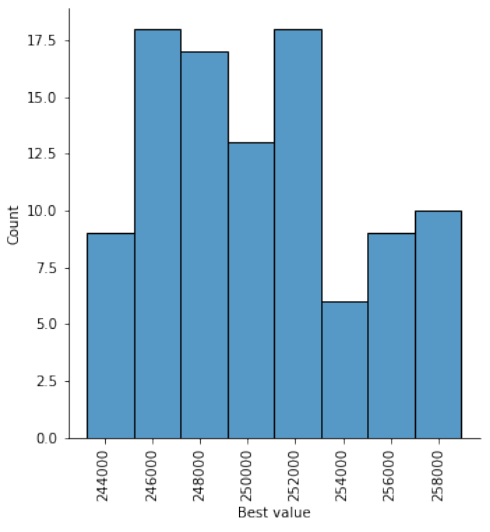


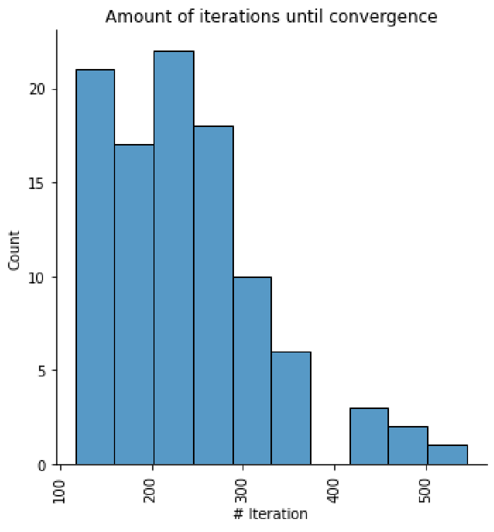


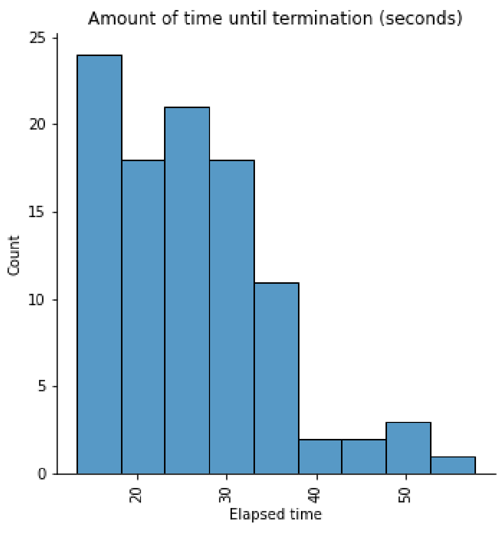
Furthermore there is relatively low variance of the best scores, they hover strongly around the global minimum. The last two plots show that the algorithm finds the optimal solution very quickly. **The one strong outlier has to to with Numba, the first iteration needs to be compiled which takes around 10 seconds.**

5.3 tour100.csv





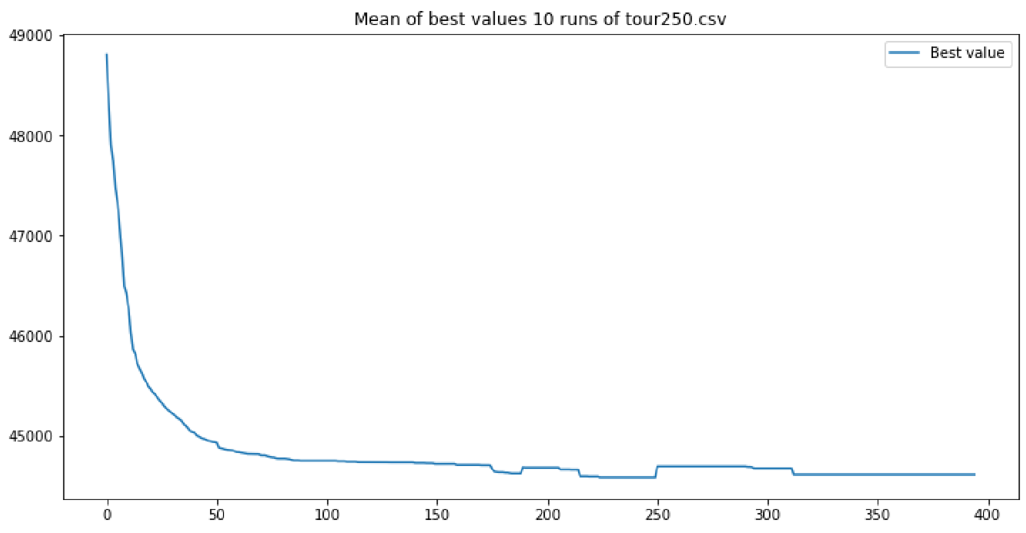


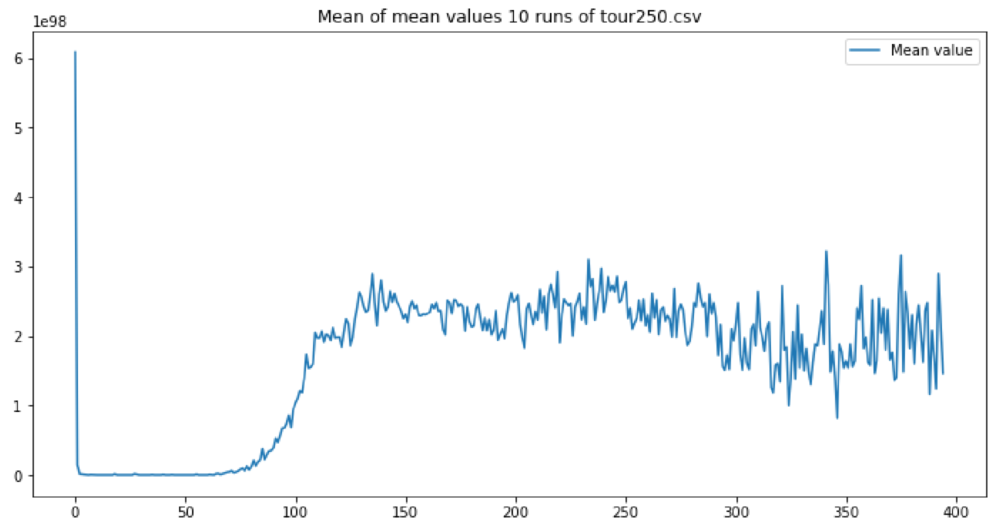


These four plots are the same as in the previous case with tour29. The algorithm is still scaling pretty well in terms of time. It can complete most of its iterations within less than 50 seconds.

5.4 tour250.csv

The best value over 5 runs was recorded at 44257.51250456195. I added this subsection to show the impact of increasing the mutation probability. With correct hyperparameter tuning the algorithm can get it to around 42000.

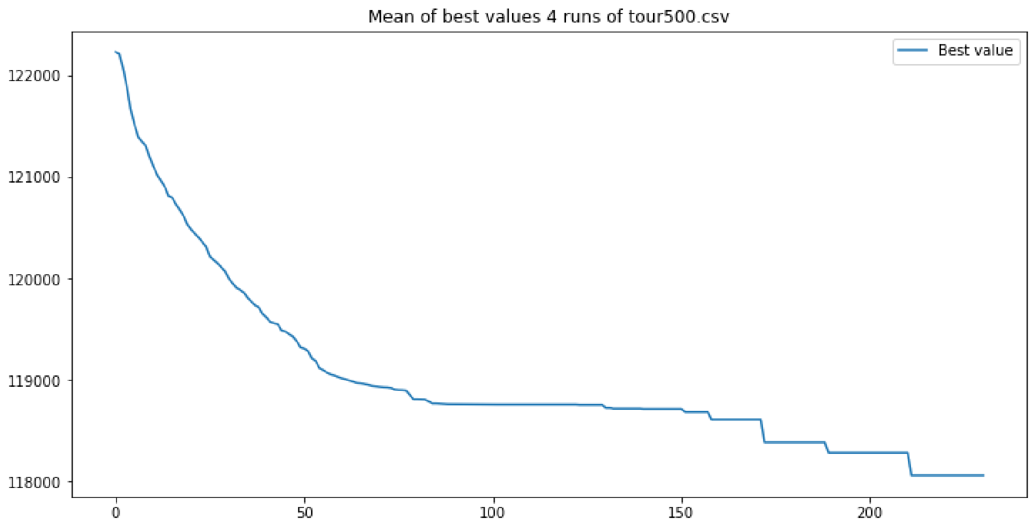


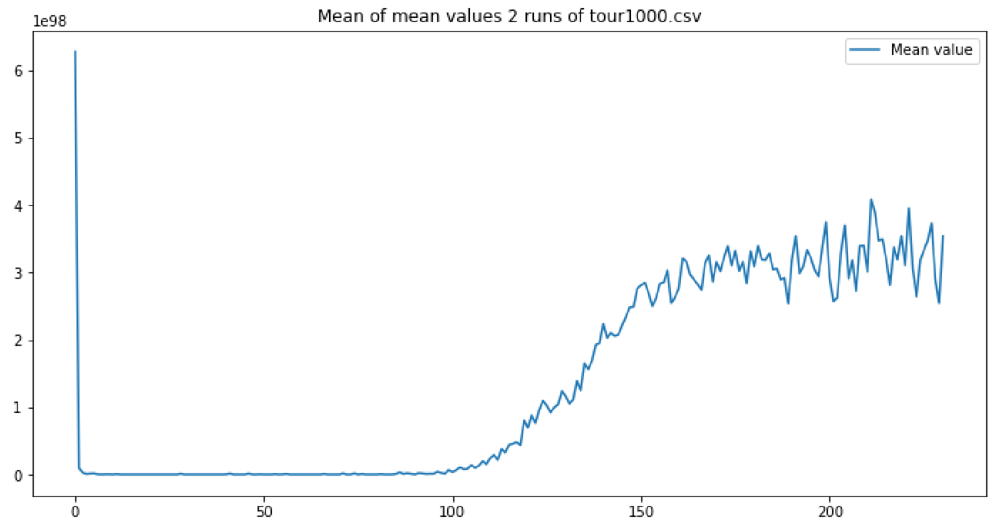


Notice how the population would normally steeply converge and increasing the mutation probability inserts enough randomness back into it.

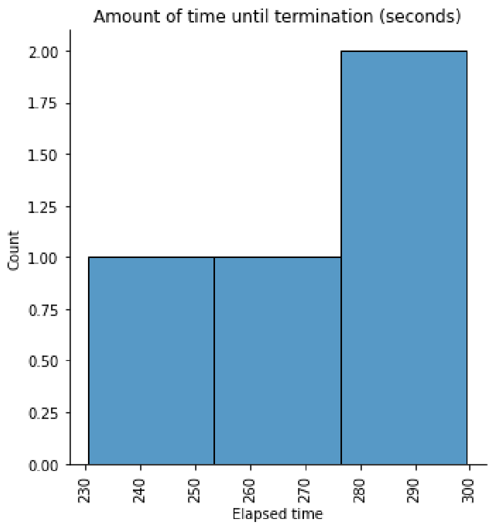
5.5 tour500.csv

The best value found after 4 iterations was 118059.43019919231.





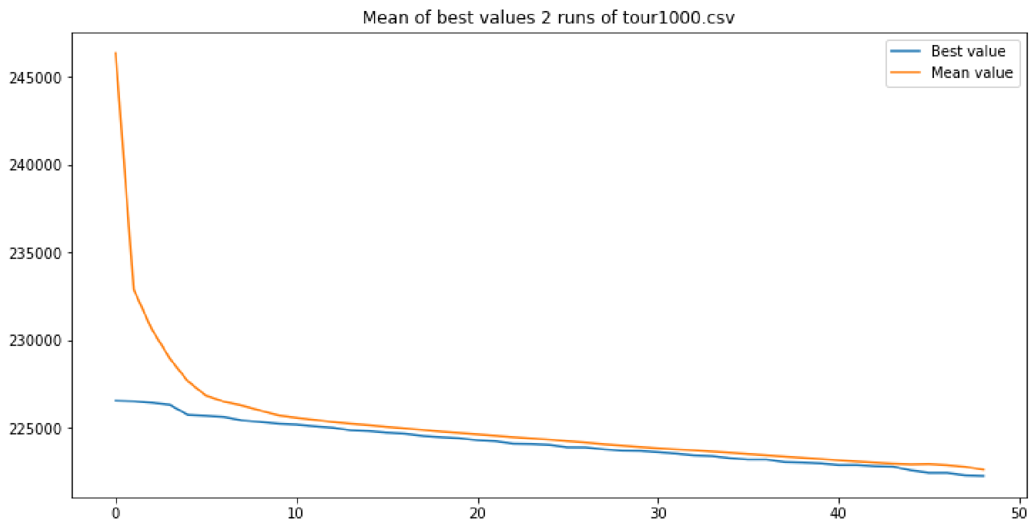
The same phenomenon occurs with as with tour250. Diversity is completely lost early on, this causes the mutation probability to slowly rise up which actually let the algorithm regain diversity and escape a local minimum. It is clear that a diversity promoting strategy would well have been in place here. The only reason why it was not implemented was for practical reasons as previously mentioned.



The algorithm still finishes running within the allotted 300 seconds as well.

5.6 tour1000.csv

Tour 1000 is where my approach is really pushed to its limits. Mainly using mutation to create variation in the population works, as shown in the other examples but is slow. This problem comes to light with tour1000.csv as the algorithm still has room to improve after 5 minutes but is stopped with a mean best value of 222264.6384951237 across two iterations.



Notice how the mean doesn't have huge spikes as as shown in tour250, this simply has to do with the fact that the algorithm is still improving its best value every iteration and hasn't fully converged yet, even though the population has lost most of its diversity.

6 Critical reflection

Strong points:

1. Suited for multi-objective optimisation problems.A lot of applications I specifically read on the topic were using NS-GA2.
2. Applicable to a lot of problems in both discrete and continuous optimisation. It's something you could apply with a lot less domain knowledge than the other two.
3. Most importantly: it is a global optimiser. With the right parameters and scheme you can find global optima for very hard problems (non-continuous functions or combinatorial optimisation).

Weak points:

1. Not efficient for cases where calculating the objective function is expensive. You could optimise everything with an evolutionary algorithm but sometimes computing the fitness over and over is not a good idea. This is probably where Bayesian optimisation shines over evolutionary algorithms.

2. Even though genetic algorithms are domain independent their variation operators aren't. Even within permutations you'd still have to figure out which operator makes sense for what use case.
3. Mainly: you're moving complexity around from one place to another. You've essentially swapped an optimisation (TSP) problem to another one, finding the optimal set of parameters for the meta-heuristic. This is especially troublesome combined with point one.

I personally think genetic algorithms are very suited for a specific set of problems. I'm happy I took the course and gotten key insight into how they work such that I can apply them later-on in my career. By implementing them from scratch I got to learn about their strengths and weaknesses in addition to where it makes sense to employ them. By reading literature and discussions from the practitioners I also know what resources to consult in the future.

The main things that surprised are firstly, how easy they are and secondly the inverse, how difficult they are as well. With the first thing I mean that it is quite easy to set up a basic genetic algorithm and get solutions that are better than a baseline you set.

On the other hand the inverse is true. Getting exceptionally good results feels like you've swapped one optimisation problem for another one: how do I set my parameters? **Even if you use self-adaptivity, the way you adapt your parameters are implicitly a hyper parameter as well.**