Evolutionary Algorithms: Final report

Vamvourellis Efstratios (r0864750)

December 27, 2022

1 Metadata

- **Group members during group phase:** Mees van der Ent and Jakub Majercik
- **Time spent on group phase:** 10 hours
- **Time spent on final code:** 60 hours (1st try) + 24 hours (retakes)
- **Time spent on final report:** 6 hours

2 Changes since the group phase (target: 0.5 pages)

1. Replaced edge crossover with **order crossover**.
2. Added **2-opt local search operator**.
3. Used **individual mutation probability** for each gene.

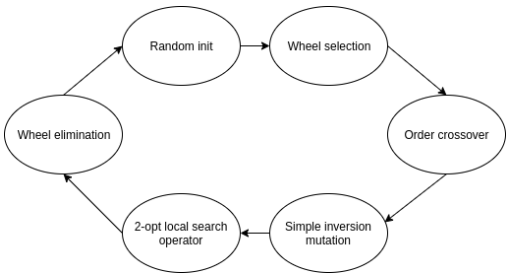
3 Final design of the evolutionary algorithm (target: 3.5 pages)

3.1 The three main features

List the three main components of your evolutionary algorithm for this project. That is, what are its most distinctive characteristics, what components am I not allowed to change to a more basic version? Ideally these are some of the more advanced features that you added since the group phase.

1. Recombination operator **order crossover**.
2. Local search operator **2-opt**.
3. **Individual mutation probability**.

3.2 The main loop



3.3 Representation

I chose the **path representation** because it was the most popular one with the most amount of recombination algorithms. A tour is represented as a list of n cities. It is the most natural representation, the tour 3-2-4-1-7-5-8-6 is represented by (32417586). The adjacency representation was considered because it is a permutation invariant representation, so we can limit our search space a lot. It wasn't implemented because we are working on the symmetric TSP which alseady has a reduced search space. The path representation was implemented as a numpy array because i am using the view feature of numpy and a few other functions that are significantly faster in numpy compared to python lists.

3.4 Initialization

I use a random initialization, **random permutations** of the cities are chosen. Some datasets have graphs that are not fully connected, this results in the majority of the initial population to be impossible solutions. To combat this, i implemented a greedy initialization method. I used a depth first search like method that tried to avoid edges with infinite cost. If the search came to a dead end and all the unvisited edges had infinite cost connections, a random edge was chosen and the search continued. This did not make it to the final version because it was very time consuming and not easy to parallelise. The random initialization, with a significant penalty to the cost function for infinite edges, and the use of LSO turned out to help clear out the population of infeasible solution in the first few generations. The population was chosen purely by experimentation. 100 seems to be a good number for the smaller datasets, whereas 200 is better for the larger ones. I settled at 200, because this small increase has minimal impact in the execution time for the smaller datasets.

3.5 Selection operators

I considered the k-tournament, **roulette wheel** and ranked based roulette wheel selection. The Ranked based roulette wheel applies selective pressure, but it is very slow, because of the need to short the genes, so it was rejected. I implemented the k-tournament and Roulette wheel. They both seem to have very similar results on these datasets. Theoretically the k-tournament should maintain a more diverse population. I ended up keeping the Roulette wheel with selection probability $\frac{fitness}{\sum fitness}$, so i don't have as many hyperparameters.

3.6 Mutation operators

I implemented the **simple inversion** and swap mutations. The mutation need to introduce enough randomness. In the symmetric TSP, the swap mutation effectively changes 4 edges whereas the simple inversion changes only 2. I think, the simple inversion works better with the order crossover recombination operator i use, so I ended up keeping it. In practice i didn't notice any difference. I use a kind of **self-adaptivity for the mutation probability**. Using a scaling function $\frac{cost-max\_cost}{min\_cost-max\_cost}$ i calculate the mutation probability of each gene. That way there is a low chance to mutate a good gene.

3.7 Recombination operators

I tried the edge, cycle and **ordered crossover**. The edge crossover seems more appropriate compared to the ordered crossover, for the symmetric TSP, as it keeps the shared edges. The ordered crossover is the fastest one using numpy views, that's why i kept it in the final,version.

3.8 Elimination operators

I considered the equivalent elimination operators to selection operators. I implemented a variation of the **Roulette wheel**, so my selection and elimination are the same. I use the roulette wheel to select the survivors, the gens that will continue on to the next generation. The only difference to the selection operator is that i dont select the same gene twice.

3.9 Local search operators

I use the **2-opt** LSO with a small twist, the max_iters hyperparameter. I limits the iterations thought the whole path. The original 2-opt iterates thought the path while improvements can be made. This is too expensive even for a medium size dataset. Furthermore, the algorithm performs only a few generations and the 2-opt heuristic, basically, does all the work. That way the other components of the genetic algorithm barely come into effect. That's why i use max_iters to limit how many times the LSO can be applied. I also considered using a random starting point for the operator, instead of it being applied at least once on the whole path. Instead i limited the amount of genes the LSO is applied to. The only genes the LSO is applied to, are the ones that have 'close to' the worst cost in the current generation, this turns out to be around 3 genes out of 200 per generation.

3.10 Diversity promotion mechanisms

I did not find the need to implement diversity promotion mechanisms since the population initialization is random and the LSO is applied a limited number of times.

3.11 Stopping criterion

I implemented a simple stopping criterion, the algorithm stops if the last $n = 10$ generations the best cost and the mean cost have barley changed.

3.12 Parameter selection

The hyperparameters where chosen purely with experimentation. Some are determined by the size of the problem and the dataset itself. For example if one edge has infinite cost, i add double the maximum distance (determined by the dataset) to the cost function. This number is found to be sufficient to apply enough selective pressure. The rest of the hyperparameters are explained in the previous sections.

3.13 Other considerations

I paralleled the LSO and selection operators. Unfortunately this was not worth it for that small of a population size and that few of LSO operations. If we decide to apply the LSO to more genes the pluralization is worth it and can be changed by the population__parallel parameter, if its equal to 2, then 2 processes will be spawned. A much better use of the 2nd core could be to use the island model, to run two different genetic algorithms in parallel, but i didn't implement that. As is, my code does not use 2 cores by default.

I used a kind of elitism thought the way the selection probability is calculated for the roulette wheel selection and elimination. The scaling function i use ensures that the best gene will not be mutated or changed by the LSO.

Sometimes the use of inherit python types was preferred over numpy for their speed, for example the numpy array distanceMatrix was transformed to a 2d tuple and most of the float operations where not done using numpy.float. I paid attention not to calculate the cost function of the same gene twice, were possible, to save time.

4 Numerical experiments (target: 1.5 pages)

4.1 Metadata

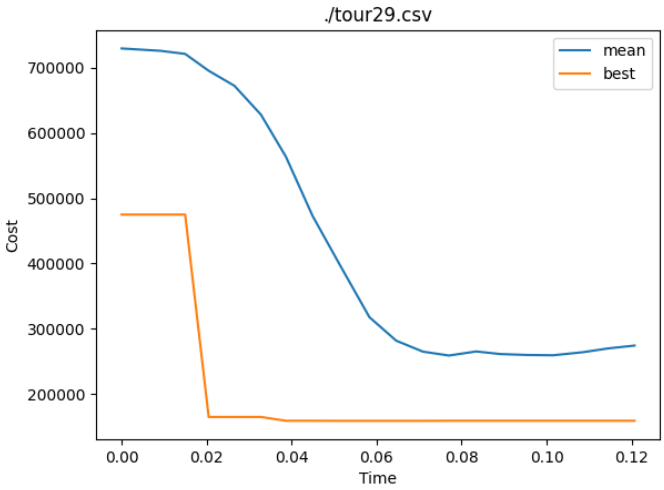
I explained my choices in Parameter selection section. All the parameters are chosen automatically, they are in the code under Hyperparameter selection, the code has plenty of comments.

My laptop has the following specs.

- CPU: AMD Ryzen 7 5800h with 8 cores, 3.2GHz (4.4GHz boost).
- RAM: 32GB.
- OS: Ubuntu 22.04 LTS
- Python: 3.8

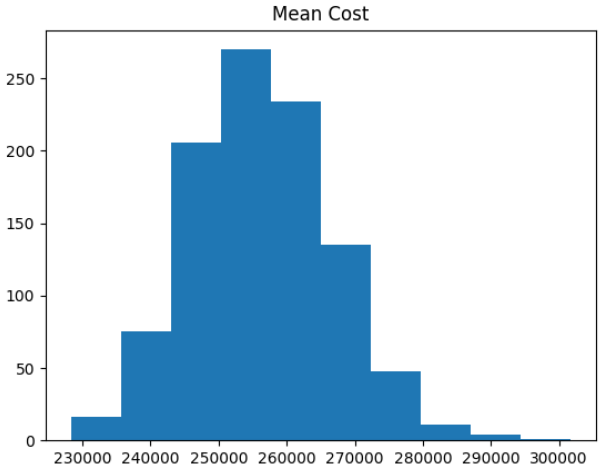
4.2 tour29.csv

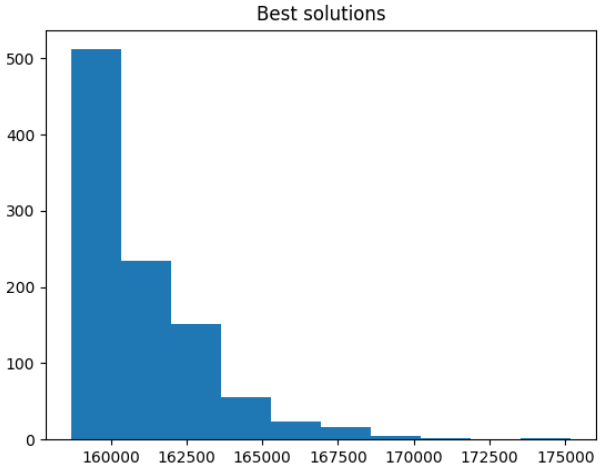
This is a typical convergence graph.



These are the results after 1000 runs.

	mean	std
time	0.1634	0.0111
best cost	160555.9216	1748.3792
mean cost	269740.1834	10191.8289
no generations	18.602	1.0486



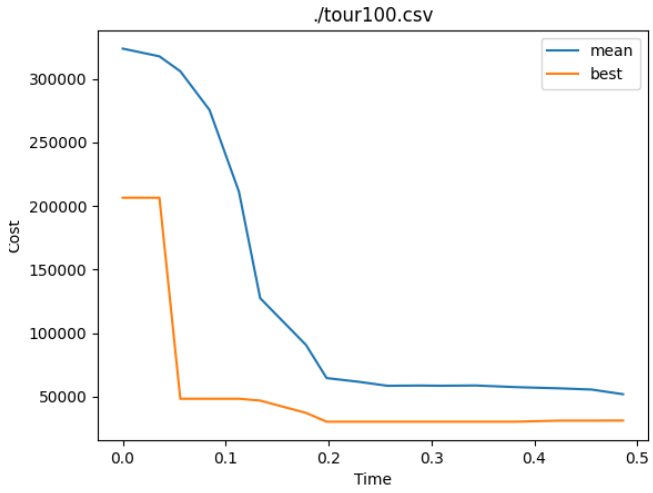


The best route was [17 26 22 21 20 28 11 25 10 6 23 1 12 18 14 13 19 27 7 0 4 8 3 5 9 16 15 2 24].

The algorithm run really fast and barley used any ram. The best solution is better that the heuristic one and the consistency is fine. The convergence is fine, considering that we need 10 generations to detect convergence and stop. Basically the best solution was found after 8 iterations.

4.3 tour100.csv

This is a typical convergence graph.

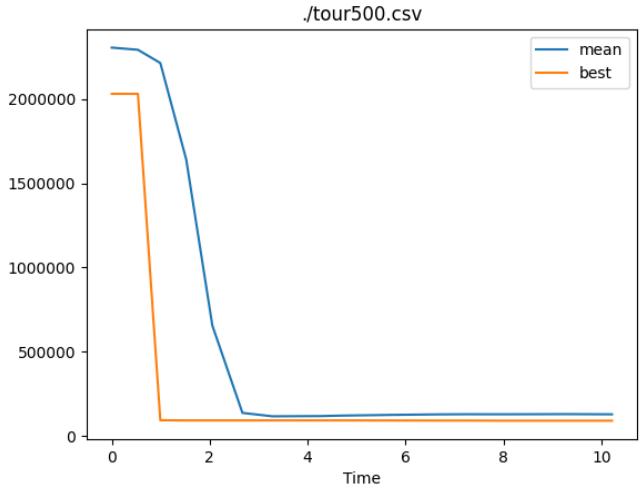


These are the results after 10 runs.

	mean	std
time	0.7488	0.0755
best cost	30513.0272	855.6493
mean cost	45496.0373	1765.6287
no generations	17.3	1.2688

4.4 tour500.csv

This is a typical convergence graph.

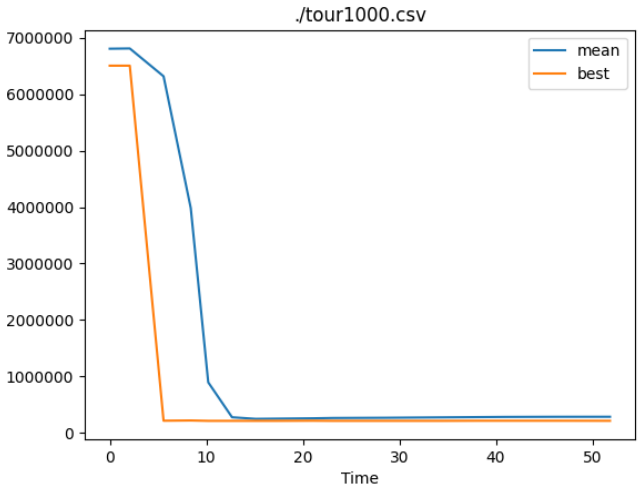


These are the results after 10 runs.

	mean	std
time	11.8780	1.4234
best cost	96895.5711	5587.4543
mean cost	136983.5612	5956.81437
no generations	16.8	1.2489

4.5 tour1000.csv

This is a typical convergence graph.



These are the results after 10 runs.

	mean	std
time	48.2876	6.1
best cost	212084.5455	4336.5124
mean cost	283500.2331	9343.6514
no generations	16.0	1.2649

We can see that the algorithm scales well. The ram used, even in this dataset is minimal. The consistency of the solutions is quite good in all datasets. I noticed that the number of generations doesn't change with the dataset size, this means that still the 2-opt LSO does a lot of the work even after all these limitations applied to it.

5 Critical reflection (target: 0.75 pages)

Main strengths of genetic algorithms

1. There components are interchangeable. You can try a lot of different variations with small changes inc ode.
2. Use simple methods that are easy to explain and understand (compared to neural networks for example).
3. They are easy to parallelise (island model).

Main weak points of genetic algorithms

1. If not designed very carefully, there results can be inconsistent.
2. They rely too much on the LSO, but are definitely an improvement over it.
3. Very hard to design correctly for large search spaces.

Evolutionary algorithms are definitely a good method to solve most NP-hard problems like TSP, especially the symmetric version, with reduced search space. I noticed that they rely to much on randomness, so you need a lot of generations an a big population to come up with exceptionally good results. Performance matters too much, that's why i think that a lower level language like C could bring much better results compared to Python. Additionally, the algorithms behind each component where easy enough to implement from scratch in any language.