Evolutionary Algorithms: Group report

Konstatinos Stefanidis - Vozikis, Pavel Mačák and Simon Merckx

December 13, 2020

1 An elementary evolutionary algorithm

1.1 Representation

We briefly considered using the adjacency representation, seeing the advantage, that each tour can have exactly one possible representation (two for symmetric TSP). The problem that we arrived at was with mutation operators, because using simple operator like swap or invert does not necessarily lead to a valid representation. We than opted for the more intuitive path representation, because of its straightforward implementation. To represent individual of our population we store the tour in a numpy array.

1.2 Initialization

We use random sampling to initialize the population. We think it would be helpful to have at least one local search operator for the TSP, which could be used during initialization, maybe after mutation or in the final stage of the evolution to really get to the best solutions. Our algorithm so far does not maintain diversity in any sense, since we do not implement any diversity-keeping mechanism, but this is definitely an area to be considered when improving the algorithm in individual phase.

1.3 Selection operators

We considered the k-tournament and roulette wheel selection. We opted for the latter since it does not involve choosing a parameter, is simple to implement and was also used in many articles on TSP that we read. However it would be worth considering to use different probability distribution f.e. logarithmic or exponential or even adapt the shape of probability function based on current iteration number etc.

1.4 Mutation operator

We considered the basic mutation operators for permutation such as swap or scramble and in the end chose to implement inverse operator on a random sub-array. We also considered implementing some kind of decreasing probability with increasing size of the sub- (when randomly choosing the endpoint), but we left this for the individual phase. It might be worth considering to implement some of the previously mentioned operators as well perhaps with different probability to introduce more randomness into our algorithm. We use base self-adaptivity by adding mutation probability to each individual, initializing it randomly from a predefined range and recombining it linearly with overlap during crossover.

1.5 Recombination operator

For recombination operator we searched through a variety of operators, that were said to work well with TSP, like edge assembly crossover, sequential constructive crossover and edge recombination crossover. Finally we opted for order crossover (OX). This operator takes some interval $< a, b >$ from first parent and puts it on the same place in the child. Then it fills the rest of the child with cities from second parent starting from index $b + 1$ (both in offspring and second parent), skipping the cities that are already present in the child. We opted for this one since it is quite simple to implement and according to some articles we read, it produces reasonably good results. The shorthand of this operator is that it puts more pressure on preserving the exact order of the cities rather than focusing on edges. However preserving order of cities also leads to preserving edges between them, so if both parents were good, this should propagate to next generation. If the parents are very different, then this operator can also introduce some random scrambling of elements from the second parent.

1.6 Elimination operators

Honestly we did not think about any drawback with $\lambda + \mu$ elimination and opted for this. However we think it would be better to implement crowding into this elimination to promote diversity in our algorithm. Preserving diversity in TSP is of utmost importance since the search space is extremely big and there are plenty of local minima that the algorithm could get stuck in, resulting in premature convergence.

1.7 Stopping criterion

For the stopping criterion we combined several criteria. The maximum number of iterations is set only for our convenience. Then, every 20 iterations it is checked if mean fitness value AND best fitness value has improved. If not, the algorithm terminates. Since a time constraint is imposed on the program run-time we could also simply end the evolutionary algorithm after the three minutes, which will probably be the case in the individual session.

1.8 Other considerations

Our algorithm does not promote diversity at this point, which should definitely change in the future if we hope for better results. We should also implement some local search operator designed specifically for ATSP and use it in initialization, after mutation and also in the final stage of the evolution to really get the best results out of diverse portfolio of tours. The positive influence of local search operator on the results was emphasized in multiple articles.

2 Numerical experiments

2.1 Chosen parameter values

One of the parameters used is population size (size of one generation) and number of offsprings. While experimenting with these parameters we kept the number of offsprings equal to population size. We arrived at the value 100, which seems optimal for the small test case. With 50 we get further away from the optimal value, while with size 500 we did not observe much improvement over size 100. For our experiments both population size and number of offsprings were set to 100.

Another parameter is the probability of mutation. In our implementation we assign a random mutation probability to our individuals when they are initialized. This probability lies between 0.01 and 0.12. We noticed that increasing this probability a small amount (to around 0.2) made our algorithm run longer. This allowed us to reach (in some of the runs) solutions with a marginally better objective value. The experiment supports our idea that the current algorithm lacks diversity. When we further increased the probability, the resulting candidate solutions became a lot worse. At the maximum probability of 1 (every individual gets mutated every iteration), the best solution we could observe had a value around 38000.

2.2 Preliminary results

As can be observed in the convergence graph (Figure 1), our algorithm manages to quickly drop the mean and best objective value of it's population. The best value encountered after 1.5 seconds is very close to the optimal solution for the problem size. However, from the graph it is clear that we do not have a diverse population. When the algorithm is running for about 1.5 seconds, the mean and best objective values start to converge and our stopping criterion is activated. The lack of diversity is most likely also the reason for the good performance of our algorithm (time, memory and convergence wise).

As seen in Figure 2, our algorithm sometime converges prematurely, i.e. before reaching good fitness value. This should also probably get resolved when diversity gets promoted. Also increasing the population size could help in providing more stable results.

2.3 List of identified issues

Lack of diversity: The main problem with our current algorithm is that it converges to the best solution in the population faster then it finds (all) possible good solutions. This implies that if the population size is not chosen big enough, the algorithm might not end up in the global optimum.

Stopping criterion: Another problem we have identified is that the stopping criterion will kick in much earlier than (close to) the global minimum if the population is too homogeneous irrespective of its size. That happens if the population lacks diversity and therefore the best value converges fast.

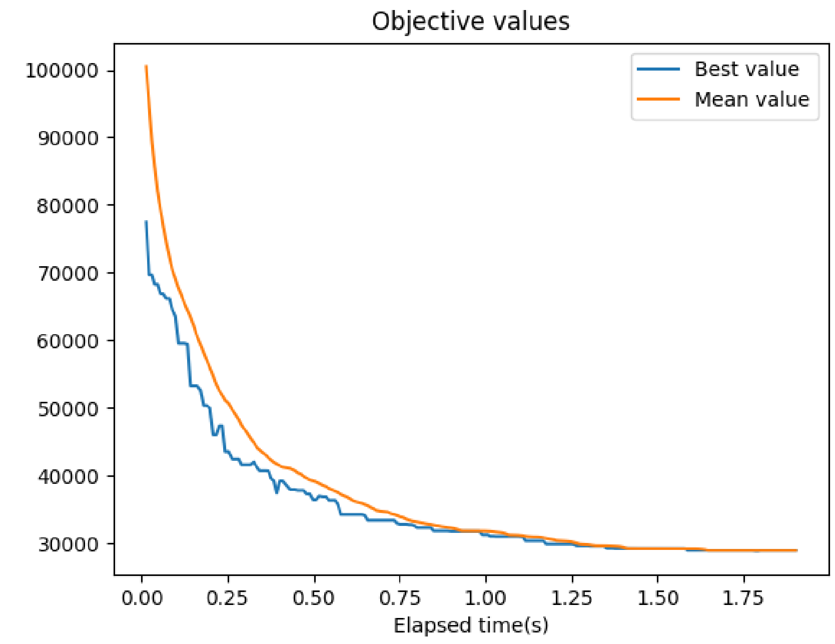

Figure 1: Best fitness and mean fitness as functions of time in seconds.

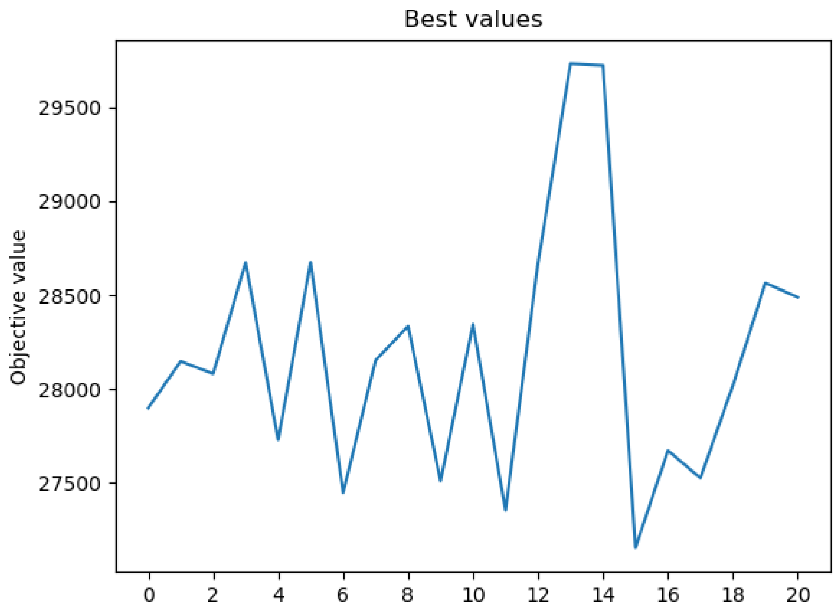

Figure 2: Best reached objective value in 20 different runs.