Evolutionary Algorithms: Group report

Konstantinos Gkentsidis, Georgios Kouros and Jeffrey Quicken

November 27, 2020

1 An elementary evolutionary algorithm

1.1 Representation

The Traveling Salesman Problem (TSP) is a widely studied optimization problem that can be solved by various optimization techniques. In this course we are interested in solving TSP with an evolutionary algorithm. The first and most important step towards this goal is deciding on how to represent the problem and its solution. The authors in [2] present five different representations for TSP: Binary, Path, Adjacency, Ordinal, and Matrix representations. Based on a review of the different methods in the same research, we decided to select the Path Representation. That's because it is one of the top methods regarding convergence on the global optimum, it is easy to implement and experiment with, and it offers a wide selection of variation operators. We implemented this representation as a python class called *Individual* that stores a list of integers representing indices of cities with respect to the given distance matrix.

1.2 Initialization

There are multiple initialization methods we can use for TSP, ranging from random permutations (more explorative) to shortest path initialization or multiple local searches (more exploitative):

- Initializing the population with Random permutations is very fast and ensures a good exploration of the search space,
- Initializing the population with a random permutation, which is then improved with a limited number of node swaps to locally improve the candidate solutions serves as another local search method aiming at exploitation,
- Initializing the population with shortest path initialization (Dijkstra's algorithm) is a local search heuristic that can insert into the population some really good candidate solutions.

To ensure a nice balance between exploration and exploitation we thought about implementing a hybrid initialization approach that involves a tunable ratio of a combination of the aforementioned methods. In the end, however, we decided to postpone the more complex initialization for the individual phase and hence in the group phase we merely used the random initialization method, which maintains sufficient diversity with a high probability.

1.3 Selection operators

Regarding selection we looked into the following operators: k-tournament selection, simulated anealing (boltzman selection), roulette wheel selection (round robin) and elitism selection(fitness based).

We decided against using a fitness-based method like elitism-based selection in order to avoid a mostly exploitative selection, resulting in a population with very low diversity that will most likely converge to a local minimum quickly and thus fail to find the global optimal solution.

In contrast we chose to implement and use the k-tournament selection algorithm that ensures a nice balance between exploration and exploitation. This selection operator can be fine tuned by selecting the best performing k. Increasing k increases the selection pressure, thus improving convergence speed, but negatively affecting diversity. This value is tuned later on as can be seen in Section 2.

1.4 Mutation operator

Mutation is another genetic operator that is applied in the process of creating a new generation in evolutionary algorithms. The mutation operator is applied to the offspring that was created as a result of the selection operations.

The mutation is probability-based and usually occurs at a low probability as it carries the risk of harming the performance of any individual it is applied to. It is used in order to prevent stagnation and ensure diversity of the population. But, if the mutation rate is excessively large, the genetic algorithm will turn into the equivalent of a random search.

In the group phase of the project, we decided to implement two mutation operations, namely the swap mutation and inversion mutation. The reason was that these mutation operators are suitable for chromosomes of ordered lists, as the new chromosome still carries a big proportion of the same genes as the original one. We decided to keep inversion mutation. This is quite reasonable if we think that inversion mutation introduces more randomness to the population in comparison with swap mutation. Finally we didn't use self-adaptivity mechanisms for the mutation, due to the fact that we wanted to keep our evolutionary algorithm simple in the group phase.

1.5 Recombination operator

Multiple recombination operators were considered for the algorithm such as: crossover, position-based crossover and ordered crossover. The recombination algorithm that was implemented is ordered crossover as described by Davis [1]. In short, ordered crossover is a permutation operator in which a swath of consecutive nodes from parent 1 drop down to the child, and the remaining values in the child are placed in the order as they appear in parent 2. This operator was chosen for its simplicity and for its proven performance in TSP problems. In a TSP problem, the fitness function is very sensitive to the relative positions of the different nodes rather than their absolute positions. It does not matter what the starting point is, it is important whether two nodes that are close together will be visited in succession of each other. Ordered crossover preserves these relative positions while switching up the absolute positions. Ordered crossover is a method that can not be controlled with parameters, since it is a simple operation. This is an advantage since it is easily understood and there is no parameter to be tuned.

1.6 Elimination operators

The requested $(\lambda + \mu)$-elimination operator is a fairly good elimination method, although it's not ideal, since it doesn't keep a good balance between exploration and exploitation, tending more towards exploitation. Keeping only the λ best individuals from the $(\lambda + \mu)$ population means that a lot of candidate solutions are eliminated because they are not as good now, although after a few generations, maybe they could yield really good offspring. As a result, we would rather use a more balanced method that keeps a part of the population based on fitness and another part in random.

1.7 Stopping criterion

Along with the provided 5-min-timer-based stopping criterion, we implemented two more stopping criteria. Firstly, we added a parameter that determines the maximum number of iterations that the evolutionary algorithm will run for. This criterion was used mostly for running experiments for a fixed number of iterations. It is controlled by a parameter called max_iters, where $max\_iters = 0$ denotes infinite iterations. Secondly, in each iteration we compute the spread of the best fitness minus the mean fitness over the N latest iterations and if it's lower than a threshold ϵ we conclude that the algorithm has converged and thus we terminate it.

1.8 Other considerations

We considered using local search mainly for the initialization step of the algorithm, so that we can get some really good initial candidate solutions from the beginning. Those are described in the initialization section of the report. We also considered implementing one of the diversity preservation methods like fitness sharing or crowding, but we decided to postpone those implementations for the individual phase of the project.

2 Numerical experiments

2.1 Chosen parameter values

The tested and chosen parameter values can be seen in Table 1. In order to decide on those values, we created various experiments for deciding the population size, mutation/recombination rate, and k for k-tournament selection. Each experiment was run independently from the other experiments using fixed values for the corresponding parameters. In the end we combined the parameters in a single experiment and fine tuned them.

Parameter	*Tested Values*	*Chosen Values*
λ (population size)	$[50, 100, 200, 500, 1000, 2000]$	2000
μ (offspring size)	$\lambda/3$	$\lambda/3$
k (k-tournament-selection)	$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$	4
α (mutation probability)	$[0.01, 0.05, 0.1, 0.2, 0.25, 0.3, 0.4, 0.5]$	0.01
recombination probabiity	$1 - \alpha$	0.99

Table 1

2.2 Preliminary results

We have run our evolutionary algorithm 20 times, for the `tour29` dataset, and we observed that the performance of our algorithm was quite good, if we consider that our implementation is quite simple. The aim of our algorithm, as we discussed before, is to find the shortest path that goes through all cities once and returns to the starting city. Therefore, the mean value of the shortest distance for the twenty iterations of our algorithm was 28340.2023 and the standard deviation was 999.0334, which is very close to the optimal value 27200. In Figure 1, we can see the evolution of our mean and best minimum fitness values during each generation, for the twenty iterations of our genetic algorithm.

It is important to mention here, that when we tried to increase the number of generations for our algorithm, the solution was not improving and it was stuck in the suboptimal solution that was reached before generation 780, as shown in the plot. From that point on, the similarity between the average value and the best minimum value indicates that this solution took over the population and therefore we will not see any improvement unless a lucky mutation turns up. This means that exploitation has overpowered exploration and we couldn't maintain the diversity for our population. Therefore, we need to introduce more exploration to our algorithm in a later stage of the individual project.

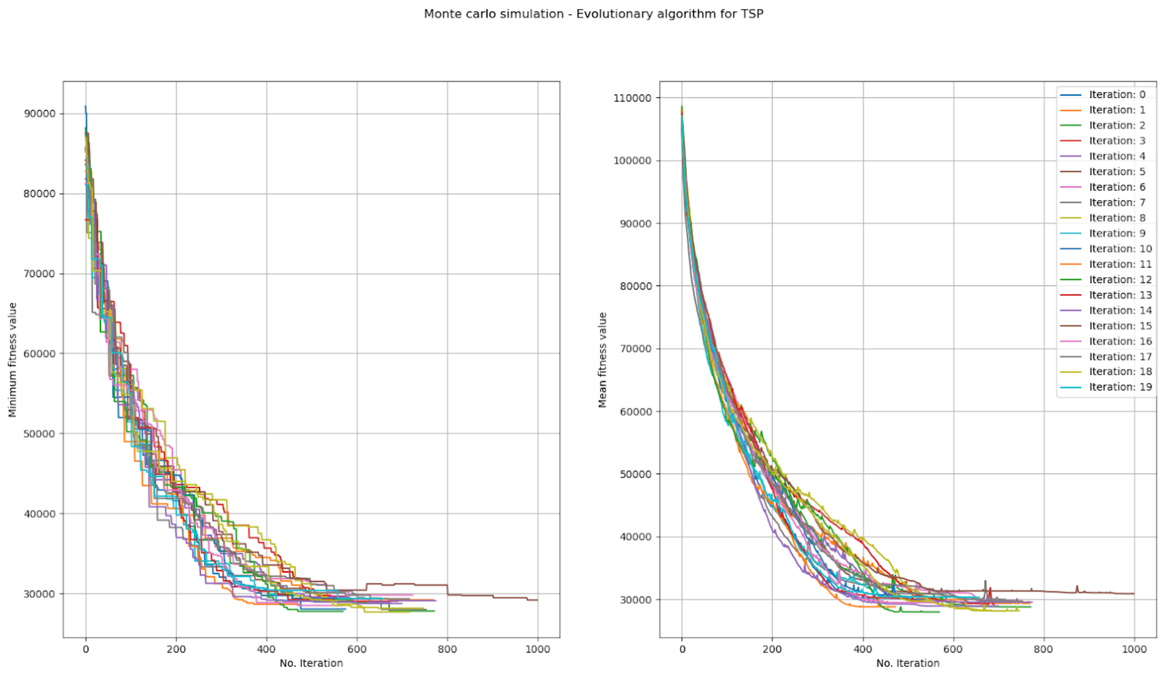


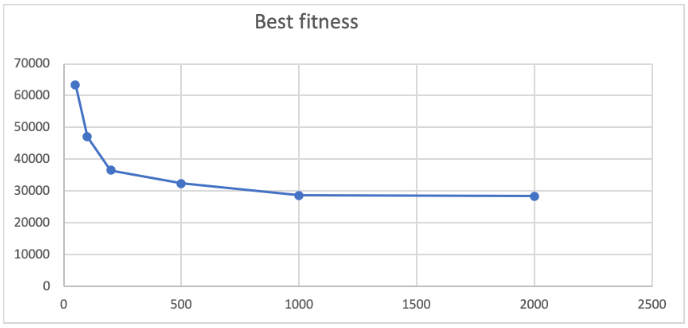
Figure 1: 20 iterations of the algorithm for tour29 dataset

2.3 Population Experiments

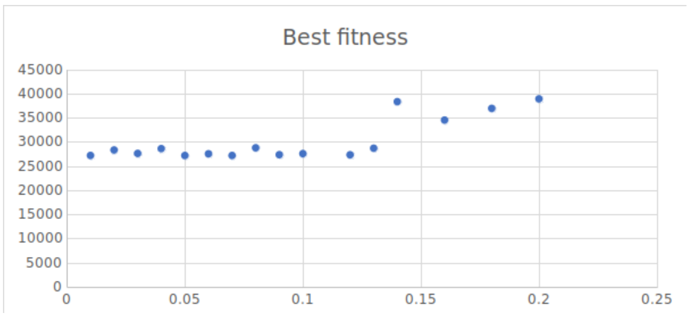
The population size was tested for the different values as described above. When examining the best fitness value (distance of solution) reached by the algorithm in Figure 2a, it can be seen that for larger population sizes, the best solution is better. For population sizes larger than 2000, the algrithm becomes slow. Therefore it is decided to use 2000 as population size for the algorithm.

2.4 Mutation Experiments

We also run some experiments for a wide range of mutation probabilities and we observed that the optimal value is 10% (see Figure 2b). Moreover, when we increased the mutation probability, the best and the mean fitness value was increasing due to the fact that our genetic algorithm was turning into a random search.



(a) Best fitness for different population sizes



(b) Best fitness for different mutation probabilities

Figure 2

2.5 k-Tournament Selection Experiments

For finding a good k value for the k-tournament selection operator we perform several runs on tour29 with constant population size, offspring size and mutation probability for different values of k between 1 and 10. Based on the produced results in Figure 3 we conclude that a value of k=4 is optimal with regard to finding a good solution, while not being too computationally expensive. We also notice that for bigger k we have a faster convergence, but our selection aimed at finding a good balance between iterations until convergence and computational complexity.

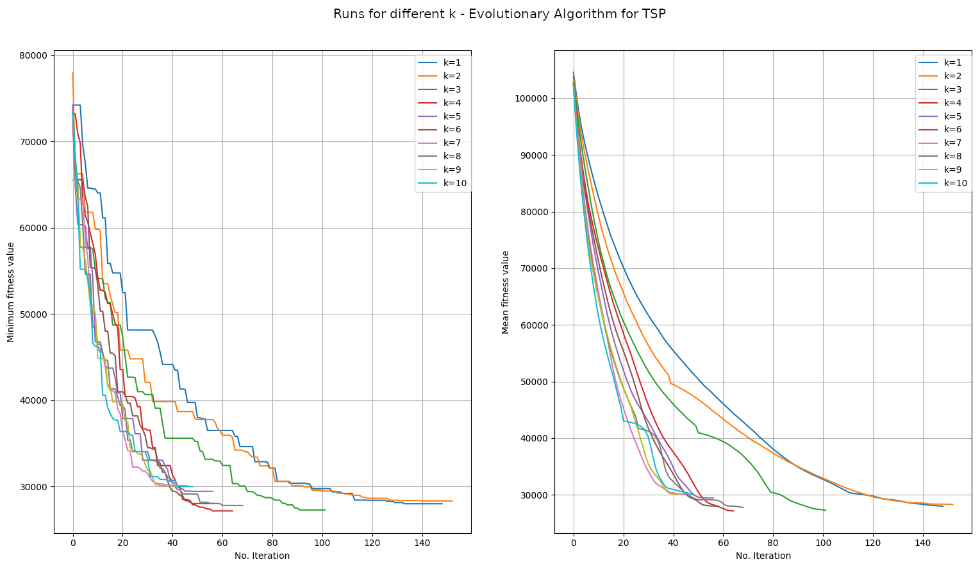


Figure 3: Best and mean fitness for different k-values

2.6 Results for all tours

In Table 1, we present the results of our algorithm for the various tour datasets of the assignment. As can be seen, we achieve good results for the smallest dataset. However, for the bigger datasets, our algorithm proves suboptimal after running for almost 5 minutes and reaching a fitness at least double the greedy hueristic in each case. For the biggest dataset in particular, our algorithm achieves a fitness value that's an order of magnitude larger than the optimal fitness and the greedy heuristic.

Tour	*Optimal Fitness*	*Greedy Heuristic*	*Our Fitness*	*Convergence Iteration*
Tour29	27200	30350.13	28793.00	76
Tour100	7350	8636.5	14720.47	297
Tour194	9000	11385.01	24450.86	321
Tour929	95300	113683.58	1986979.45	31

Table 2

2.7 List of identified issues

- The algorithm doesn't preserve diversity well. We see convergence after a few tens/hundreds of iterations. In order to solve this problem we could look into diversity promotion methods, keep a low enough k in k-tournament selection and/or implement a more exploration-friendly elimination operator.
- The convergence criteria we used often cause the algorithm to finish early. This can be observed if the criteria are removed and the algorithm is left to run for eg. 10,000 iterations. Then there are long periods of stagnation, until the fitness finally improves. This can be explained by the randomness of the mutation and recombination operators. When the algorithm has found a really good candidate solution, it's really difficult for the right variation to occur that improves upon said solution.

3 Other comments

References

[1] L. Davis, *Applying adaptive algorithms to epistatic domains*, in Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1, IJCAI'85, San Francisco, CA, USA, 1985, Morgan Kaufmann Publishers Inc., p. 162–164.

[2] A. Hussain, Y. S. Muhammad, M. N. Sajid, I. Hussain, A. M. Shoukry, and S. Gani, *Genetic algorithm for traveling salesman problem with modified cycle crossover operator*, Computational Intelligence and Neuroscience, 2017 (2017), pp. 1–7.