Evolutionary Algorithms: Group report

Alexander Lohaus, Guoshuo Ye, and Taofeng Bu

November 9, 2023

1 A basic evolutionary algorithm

1.1 Representation

To represent the different elements of the problem-solving algorithm, 3 classes are used that each represent a different part of the problem: The **Parameters** class is a wrapper class that contains the parameters for the problem-solving stage, such as population size and amount of iterations. The second class, **TravelingSalesmanProblem**, describes the problem itself. It contains information about the number of cities and the distances between these cities (encoded in a distance matrix). It also contains a variable `params` which is a **Parameters** class containing the parameters used in the problem-solving process. The final class, **Individual**, is a representation of a single solution. It contains a list `order` which describes the order in which the cities are visited. It also contains a parameter α which indicates how likely the individual is to mutate itself in each iteration.

1.2 Initialization

For the initialization, the random initialization method is used as it is recommended as a default choice. The initial population size is controlled using the parameter λ. Random combinations of different representations of cities are generated in lists and each list contains a random combination of 100 cities without duplication. The approach ought to cover the domain well in this way.

1.3 Selection operators

For the computation of fitness, the total distance of the selected cities is computed, and the selection of path is based on the fitness function.
For the selection part, the competition-based selection—k-tournament selection method is implemented within our genetic algorithm to solve the traveling salesman problem. A specified number of individuals, denoted by k, are randomly chosen from the population without replacement. This selection is carried out such that each individual has an equal chance of being chosen for the tournament, ensuring fairness in selection. Among the selected individuals, the fitness values, which correspond to the total distance of the tour, are evaluated, and the individual with the lowest fitness value is chosen to move on to the next generation. The process of k-tournament selection is combined with recombination operation to generate the offspring for the next generation. By doing so, it allows for the preservation and propagation of good solutions through generations while maintaining a diverse range of solutions.

1.4 Mutation operator

A swap mutation operator is chosen in our evolutionary algorithm, by randomly mutating an individual by swapping the order of two random points. Swap mutation is suitable for our representations of the Traveling Salesman Problem, as the cities of each individual are represented as an ordered sequence. To maintain the balance between exploration and exploitation, the swap mutation operator only makes a small change in the order of city visiting of an individual. Otherwise, it is easy and straightforward to implement and has low computational overhead. We don't evaluate to what extent randomness our swap mutation operator introduced solely. Overall, our evolutionary algorithm would not converge very quickly, which shows our variation operators do introduce sufficient randomness. A parameter α controls the individual mutation rate which is an attribute of the Individual class.

1.5 Recombination operator

The recombination of two parents into a child is done according to an algorithm known as *Order Crossover*. The idea is to initialize an empty offspring: an array with the same length as the parent, with all values initialized to `None`. Then, the middle part of the first parent is copied to this offspring. Next, the algorithm loops through all

cities in the second parent (in order) and copies the cities from the second parent to the offspring if they do not yet exist in the offspring. This results in a new order of cities that is new, but still contains some characteristics from both parents.

1.6 Elimination operators

An $(\lambda + \mu)$ elimination operator is implemented in our evolutionary algorithm. It combines both parents and offspring and calculates all fitness values. All individuals are ranked by their fitness values. The top λ individuals are retained. The only parameter here is the population size λ. The $(\lambda + \mu)$ elimination strategy can maintain a higher diversity with both individuals from parents and individuals considered for the next generation.

1.7 Stopping criterion

For our parameter search, we let the algorithm run for 60 seconds each time. From experiments, it became apparent that this gave the algorithm enough time to get an idea of its solving behavior and speed.

2 Numerical experiments

2.1 Chosen parameter values

λ (Population Size)	μ (Offspring Size Ratio)	k (Tournament Size)
250	0.5	3
1000	1	5
4000	2	10

Table 1: Parameter values for λ, μ, and k

For the parameter value, we chose 3 values for each of the λ, μ, and k to analyze the use of each value here. We define μ using the ratio depending on the population size.
We have observed that a larger population size increases the diversity of solutions that the algorithm explores, which can help prevent premature convergence to suboptimal solutions but also increases computational complexity because more candidate solutions need to be evaluated; A larger offspring size allows the algorithm to explore a wider range of potential solutions in each generation, which can help to speed up convergence. However, similar to population size, a larger offspring size also requires more computational resources; A larger tournament size tends to favor individuals with better fitness, accelerating convergence towards optimal solutions. However, excessively large tournament sizes can reduce population diversity and lead to premature convergence to suboptimal solutions.

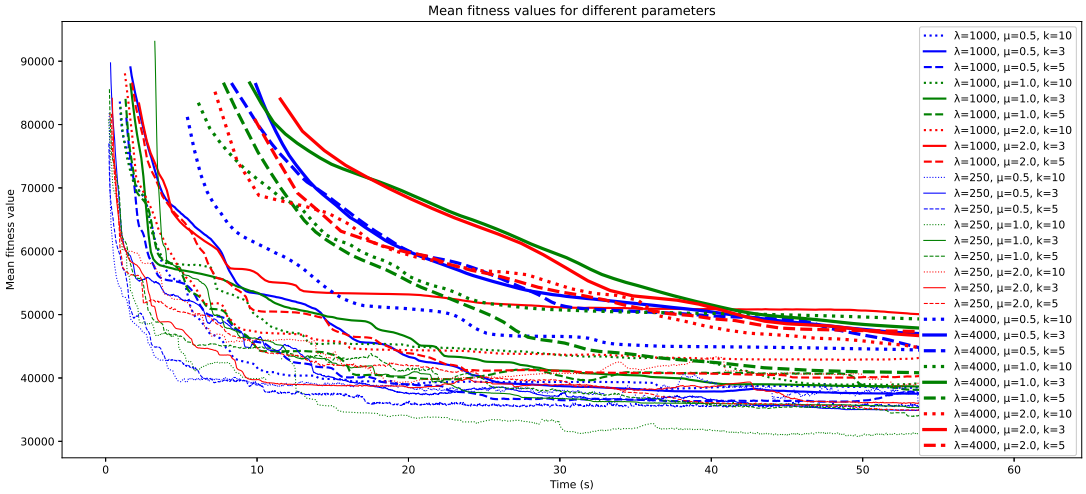

Figure 1: Mean fitness values for different parameters

After running multiple experiments with a variety of parameters, it is clear from Figure 1 that some choices will heavily impact the performance of the algorithm. First, it's apparent that increasing the population size past a certain point has a negative effect on performance. It seems the additional exploration in each iteration is not

worth the extra time it takes to complete each iteration. Increasing the tournament size accelerates algorithm convergence but reduces exploration. However, after 1 minute the reduced exploration does not seem to negatively affect the final solution of the algorithm. Finally, the different tested offspring sizes seem to have a negligible impact within this time frame.

It is important to note that due to the randomized nature of this algorithm, a combination of parameters might do very well in one specific run but might do worse in another. For example, in Figure 1 the combination of λ=250, μ=1.0, and k=10 seems to be very successful but after rerunning the experiment multiple times, we found that it did not significantly outperform other parameter combinations as it did in the first run. It's possible that in this specific run, the algorithm simply got lucky with a certain random guess and converged further into this path.

2.2 Preliminary results

For the final run of our algorithm, we will choose the parameters as follows: λ=250, μ=1.0, and k=5. Figure 2 is the result of running the algorithm for 5 minutes, where the final mean and best value are 28265 and 25699 respectively.

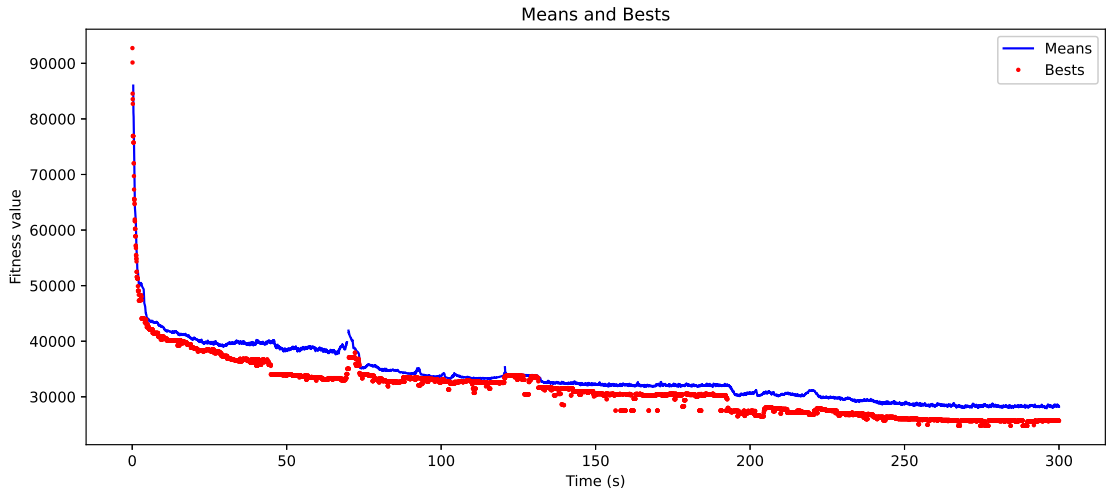

Figure 2: Convergence plot (means and bests) over time

After running the algorithm 50 times for 1 minute, a variation in the final solution becomes apparent. The variation is shown in the boxplot in Figure 3.

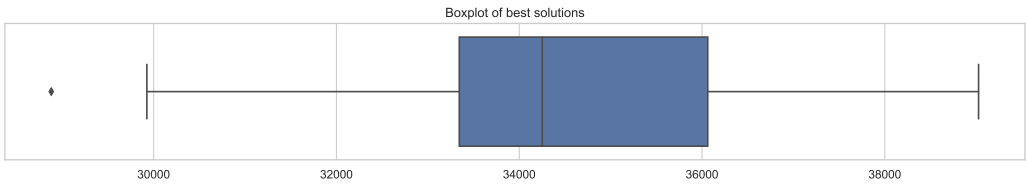

Figure 3: Boxplot showing the variation over 50 runs of the algorithm

The result of our algorithm running for 5 minutes is reasonable for an evolutionary algorithm on a problem of this size. As seen in Figure 2, the algorithm finds improving solutions fairly quickly at first and continues to make slow improvements over the full 5 minutes. This steady progress demonstrates a relatively well-balanced mutation and selection process, preventing stagnation in local optima and promoting continuous exploration of the search space. In terms of factors like these, we believe the performance of our algorithm is reasonably good, especially considering its general-purpose evolutionary approach rather than using specialized TSP knowledge.

However, it is unlikely this is the true global optimum, as evolutionary algorithms cannot guarantee finding it, especially in a limited run-time. The landscape for TSP problems is complex with many local optima. While we found good solutions, there is a chance an even better tour exists that our algorithm did not discover within 5 minutes. Trying a better combination of operation methods or running for longer may find an even better solution but would be more computationally expensive.