Evolutionary Algorithms: Group report

Kaixi Yao, Sarthak Janjuha, Julius Jacobitz and Matties Roofthooft

November 4, 2025

1 A basic evolutionary algorithm

1.1 Representation

A candidate solution is represented using a permutation of all cities except for the fixed starting and ending city. Therefore, a candidate solution is represented as a permutation of the $N-1$ cities from 1 to $N-1$. By using a permutation, we ensure that every solution our algorithm creates is a valid tour. We simplify the problem since it only has to optimize the 49 "in-between" cities. We considered the classic binary representation but quickly dismissed it. Encoding a city sequence into a string of bits would be highly complex. Each candidate solution is stored as a 1D NumPy array of integers. The population is a 2D NumPy array with dimensions (`POPULATION_SIZE, N-1`). For example, `population[i]` gives the i-th individual in the population, which is a 1D array representing its tour.

1.2 Initialization

We initialize the population by generating a set of completely random permutations. Each individual in the starting population is a random, unique ordering of all the cities (nodes 1 to $N-1$). We first create a list containing all the cities that need to be visited, which is $[1, 2, 3, ..., N-1]$. We then use NumPy's `rng.permutation()` function to randomly shuffle this list. Checking each individual for length $<$ inf results in random feasible tours where each city appears exactly once.

1.3 Selection operators

We considered two selection operators: fitness-proportionate selection (roulette wheel) and tournament selection. We choose k-tournament selection because k directly controls the selection pressure and it works well even when fitness values have a large range (like our tour lengths), as it only compares relative fitness within the tournament group. Small k (e.g., 2-3) means a more diverse selection, weaker individuals have a better chance. Large k (e.g., 5-10) means a more aggressive selection, strongly favoring the best individuals.

1.4 Mutation operator

We considered the following mutation operators: swap mutation, inversion mutation, scramble mutation. Based on the prompts, the inversion mutation is set as the default scheme. It effectively acts as a "macro-mutation" that can create a significant change in the tour structure while still preserving many adjacent city pairs (or "edges"). It also introduces a powerful form of randomness. We use a dynamic mutation rate that starts at a base level (e.g., 0.3) and increases when the algorithm stagnates, encouraging more exploration. Besides, the potential size of the change is inherently controlled by the segment size, allowing for both small tweaks and large-scale rearrangements.

1.5 Recombination operator

We considered the following crossover operations:

- Cycle Crossover (CX): Identifies position cycles between parents and swaps them alternately, ensuring each gene inherits its position from one parent while maintaining valid permutations.
- Order Crossover (OX): Copies a subsequence between two cut points from one parent and fills remaining positions with genes from the other parent , preserving relative order.
- Position-Based Crossover: Randomly selects positions from one parent to copy into the offspring; remaining spots are filled with genes from the other parent in order, avoiding duplicates.

We are using Ordered Crossover (OX) as it is easy to implement for our use case and provides us with good results. It preserves the relative order of cities between parents, and we want to promote a healthy order between generations.
Using OX, we can preserve partial sequences or subtours, which can represent locally optimal routes. If the two parents have little overlap, the fixed segment taken from Parent A still transfers one good partial structure. The

remaining sequences filled from Parent B will likely produce high diversity offspring, potentially exploring new parts of the solution space.
We can control the OX behavior via the choice of cut points (two random indices). By adjusting how these cut points are chosen, we can control how much genetic material is inherited from each parent. Shorter segments lead to less parental inheritance, and more exploration and longer segments lead to more exploitation. Currently in our algorithm, we are using a random length substring. The OX crossover is static in our implementation.

1.6 Elimination operators

We considered $(\lambda + \mu)$ and (λ, μ) elimination operators, and chose (λ, μ)-elimination with some Elitism, so the best members are directly copied to the next generation. It guarantees that the best solutions are never lost.

1.7 Stopping criterion

We combined two stopping criteria for the algorithm. The primary criterion is a **maximum generation limit**, which serves as a tunable hyperparameter control when the algorithm terminates. Additionally, we implemented a **soft adaptive mechanism** through a self-adaptive mutation rate that increases when no improvement is observed over several generations. If no improvement is seen for a predefined number of generations (e.g., 300), the algorithm terminates early.

2 Implementation generation

2.1 Prompts to generate code

We wrote a single prompt consisting of the context information and a description of each component of the genetic algorithm. The result was already pretty good, with a little tweaking the method performed already pretty well. We inserted another prompt asking for a set of new mutations which copilot did implement correctly. Our prompts can be found in appendix B.

2.2 Prompts to fix errors

The code was free of coding errors, but the parameter selection was off. The code was manually adjusted to fix input/output related tasks.

2.3 Critical reflection

Providing the AI with contextual information significantly improves the generation of the code. For a basic evolutionary algorithm, the results are astounding. Copilot produced readable and error-free code which makes it easy to manually tweak if necessary. A risk to bear in mind is that using AI as the baseline of your program could restrict the extendability of the code. The use of AI tends to be more efficient than writing code from scratch provided that you carefully design a prompt and read the generated code to ensure errors are avoided are corrected.

3 Numerical experiments

3.1 Chosen parameter values

3.1.1 Parameters

Population size, **Number of generations**, **Selection method** (tournament selection with tournament size k), **Crossover rate**, **Adaptive mutation rate** (base value, maximum value, and increase factor), and **Elitism** (number of elite individuals to keep).

3.1.2 Parameter Selection Flow

In general, the parameter selection flow was an iterative approach changing parameters based on previous knowledge from the lectures and educated guesses. Letting the algorithm run with a given set of parameters, evaluating the results and optimizing parameters in a specific way to change the behavior of our algorithm to provide better results. For example, decrease selection pressure to converge slower.
We started with educated guesses on initial parameters, comparing the results against the given baseline. We immediately increased "Max. Generations" No improvement generations to let our algorithm explore longer.
We experimented a lot with different values and value combinations to fine-tune the behavior of our algorithm:

- Decreasing population size helped us to speed up the process and pave the way for local optimization techniques (which scale linearly with population size)

Parameter	Value
Population size	200
Max generations	5000
No improvement generations	300
Selection method (tournament k)	5
Adaptive mutation rate (base)	0.3
Adaptive mutation rate (max)	0.9
Adaptive mutation rate (factor)	1.05
Elitism (number of elite indiv.)	2

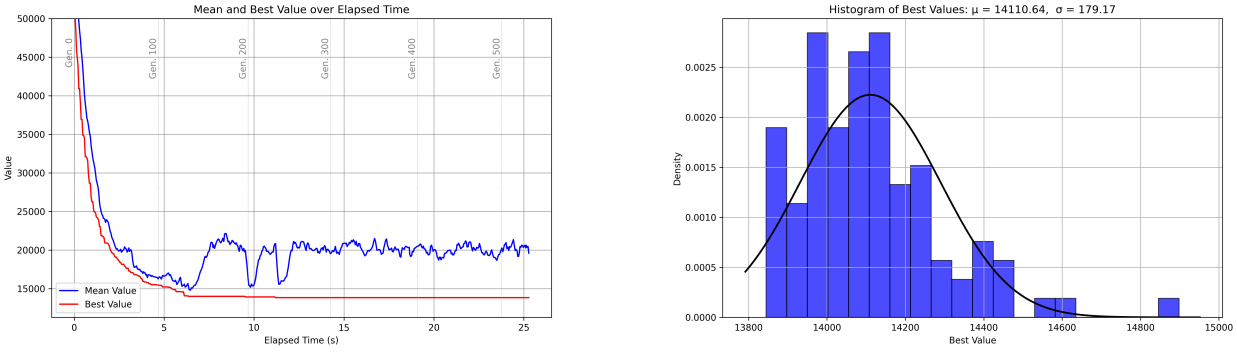
Table 1: Chosen parameter values for the evolutionary algorithm

- Because we had the feeling that our algorithm didn't do enough exploration, we changed mutation rates, k-tournament selection rounds and elite counts.

In the end Table 1 lists the parameters that seemed to return the best results, surpassing the baseline.

3.2 Preliminary results

In total we executed the algorithm with the chosen parameters from Table 1 on 100 different random seeds. Figure 1a shows a convergence graph of the best run. The **best value** we achieved was **13844.3383**. We do not think this is the global optimum, as there is no proof of optimality and there is always a slight chance that a better solution exists.
When solving this problem on 100 different random seeds, we observed some variation in the best solution. Figure 1b shows a histogram of the best values found over the 100 runs. It shows a mean of 14110.64 with a standard deviation of 179.17. This is definitely a significant variation, indicating that the algorithm's performance is still sensitive to the initial random seed. (But at least its a little scewed towards lower values). Figure 2b shows the convergence of all 100 runs.
The program ran for for apprximately 17-20 seconds until convergence and stops after 300 generations of no improvement. This is not the fastest runtime but we wanted to use the given time of 5 min to find a very good solution. It converges relatively quickly (within 300-400 generations). Figure 2a shows nicely how the adaptive mutation rate forces exploration after some time of stagnation. Memory usage is not optimized but should be fine for all problem sizes as population management and individual representation are done efficiently. The mean value for the whole population being way higher then the best member could indicate a high population diversity (Figure 1a).



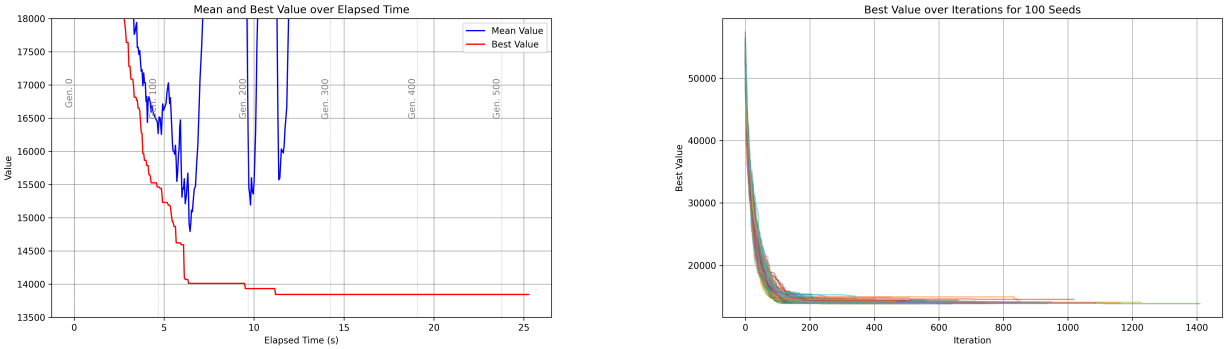
(a) Best run convergence graph over time.

(b) Histogram of best values for 100 different random seeds.

Figure 1: Comparison of convergence over time and distribution of final best values.

Appendix

A. Additional Figures



(a) Zoomed in version of Figure 1a.

(b) Best value for 100 different random seeds.

Figure 2: Detailed visualization of convergence behavior across runs.

B. Prompts

prompts

We are solving the travleing salesman problem on a graph with an genetic algorithm. We are looking for the shortest path between nodes. Your data in the given csv is a graph representation by a nxn matrix with distances between the nodes in each cell. If 2 cells are not directly connected, they have a distance of infinite. Its a directed graph.
You have the given code template.
please generate each step of the following exercise:
Generate a representation, the objective function „length", a simple population initialization, a selection mechanism, one mutation operator, one recombination operator, an elimination mechanism, and a stopping criterion that corresponds to the design that you developed in the first exercise session
representation:

- a possilbe path representation would be a list with all nodes visited in the circle. As the starting (and endpoint) are always fixed, we can just ignore vertex 1

Initialization:

- start with random population of possible paths (easy).
- it could be worth it to test for feasability (length != inf.) at initialization

Selection:

- decide on a basic way to do this - we thought about k tournament selection

Variation:

- Mutation: All basics for permuations
- Combination: order crossover

elimination:

- some form of elitism (always keep top 5 percent)
- besides that decide on a strategy

---

add Insert mutation Swap mutation Inversion Mutation Scramble mutation as optional mutation schemes

---

add dynamic mutation rate to this evolutionary algorithm