Evolutionary Algorithms: Group report

Group Member 1, Group Member 2, 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. The result is a random, feasible tour where each city appears exactly once.

1.3 Selection operators

We considered two selection operators:

- Fitness-Proportionate Selection (Roulette Wheel): where individuals are selected with a probability proportional to their fitness.
- Tournament Selection: where a small group of individuals is randomly selected, and the best one from that group is selected.

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. We use $k = 5$ since it creates sufficient selection pressure to quickly exploit good solutions while still maintaining enough diversity to avoid premature convergence.

1.4 Mutation operator

We considered the following mutation operators:

- Swap Mutation: randomly selects two positions in the tour and swaps the cities at those positions.
- Inversion Mutation: randomly selects a contiguous subtour and reverses the order of the cities within it.
- Scramble Mutation: randomly selects a contiguous subtour and completely shuffles the order of the cities within that segment.

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 Ordered Crossover, 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. We have kept the elitism rate as 5%.

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, helping the algorithm escape stagnation and maintain search diversity.

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.

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

What parameters are there to choose in your basic evolutionary algorithm? How did you approach parameter selection for now? Did you experiment a bit with different values? What was the influence of this choice? Which are the parameters that you will use for the experiments below?

3.2 Preliminary results

Run your algorithm on the smallest benchmark problem. Include a convergence graph, by plotting the mean and best objective values in function of the time. How long did your program run until convergence? How do you rate the performance of your algorithm (time, memory, speed of convergence, diversity of population, etc)? What was the best fitness value you found? Do you think this is the global optimum?

When you solve this problem several times, how much variation do you observe in the best solution? Include an informative plot of this.

Appendix

A. Example code snippet

Below is an example of a core function used in our evolutionary algorithm implementation.

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

B. Additional results

Figure 1 shows an additional run of the evolutionary algorithm on a different benchmark instance.

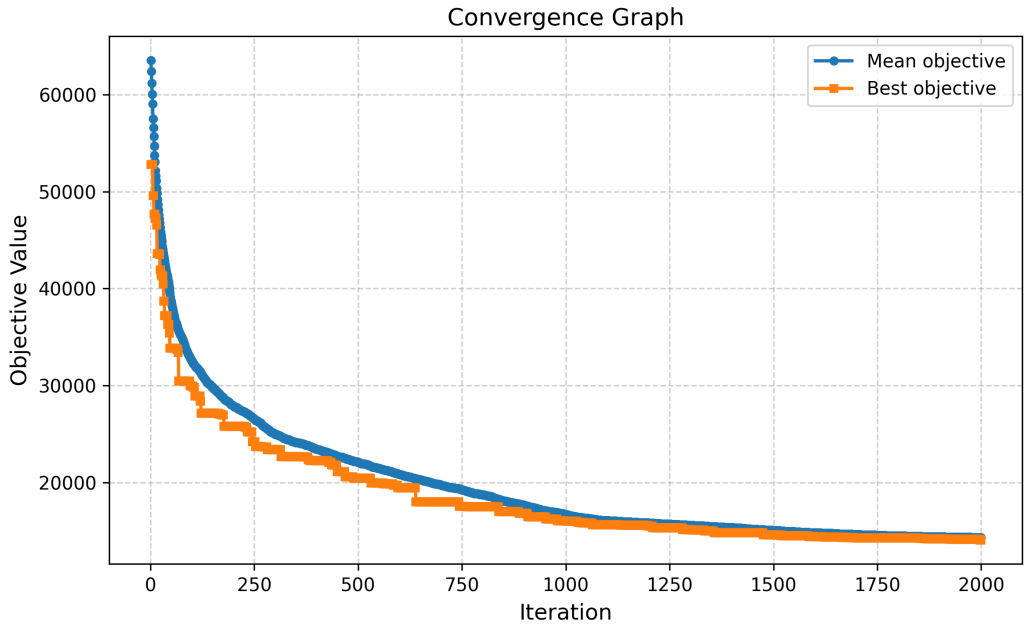

Figure 1: Convergence of mean and best fitness values over generations for an additional test case.