Exercise Report

Course: 06. Aprendizaje automático (Machine Learning): conceptos, metodología, algoritmos de aprendizaje para analítica descriptiva, predictiva y prescriptiva, y retos en su implementación

Exercise Title: Homework 1.B- Optimization (advanced, non-mandatory).

Submitted by: Jorge de la Torre García (DNI), Lydia Phoebe Amanda Lilius (DNI), Miguel Galán Cisneros (DNI), Vitor Oliveira de Souza (Z0963220P).

February 26th, 2023.

Introduction

Overview of TSP

The Traveling Salesman Problem (TSP) stands as one of the most studied and seminal problems in the field of combinatorial optimization. At its core, the TSP seeks to find the shortest possible route that allows a salesman to visit a set of cities exactly once and return to the origin city, thereby minimizing the total travel distance. This problem is not merely academic; it mirrors real-world challenges in logistics, supply chain management, route planning for delivery services, and even in the layout of electronic circuitry. The complexity of TSP lies in its exponential growth of possible solutions with the addition of more cities, categorizing it as NP-hard and making exact solutions increasingly impractical for large numbers of cities.

Objective

The primary aim of this project is to use the power of genetic algorithms (GAs) to develop a robust solution for the TSP. Genetic algorithms, inspired by the principles of natural selection and genetics, offer a meta-heuristic approach to search and optimization problems. By simulating the process of natural evolution, GAs iteratively refine a pool of candidate solutions towards an optimal or near-optimal solution. This project specifically focuses on:

Designing and Implementing a Genetic Algorithm: Developing a GA capable of efficiently solving TSP instances, with a focus on problems TSPLib repository and **2D spaces**.

1. Fitness Evaluation: Crafting a fitness evaluation function that accurately computes the total distance of a given route, facilitating the effective selection and evolution of candidate solutions.
2. Optimization and Analysis: Through iterative testing and refinement, leveraging from the concepts of Grid Search and Cross validation, optimizing the GA's performance to yield high-quality solutions that are close to the known optimal routes for the chosen TSP instances.

Methodology

Data Source

Our investigation into the Traveling Salesman Problem (TSP) uses the comprehensive collection of benchmark instances provided by the TSPLib repository, a widely recognized resource in the field of combinatorial optimization. For the purpose of this project, as suggested by the task description, we initiated our experiments with the berlin52 instance. This choice was motivated by its moderate size, allowing for manageable computational complexity while still presenting a significant challenge for optimization algorithms. The berlin52 instance represents a real-world problem of finding the shortest possible loop through 52 locations in Berlin, making it an ideal starting point for testing the effectiveness of our genetic algorithm.

Parser Implementation

To facilitate the use of TSPLib instances, we developed a parser capable of reading and interpreting the unified format in which these instances are provided. This format typically includes the list of cities, each represented by a unique identifier and its coordinates (either in 2D or 3D space). Our parser is implemented in Python **only for 2D spaces** and extracts this essential information, transforming it into a structured format that our genetic algorithm can easily process.

Development of an Extensive Genetic Algorithm Library

In the process of addressing the Traveling Salesman Problem (TSP) with genetic algorithms, we recognized the versatility and power of genetic algorithms in solving a wide range of optimization problems. Motivated by this insight, we embarked on developing an extensive genetic algorithm library.

This library is designed not only to solve the TSP addressed in this report but also to provide a robust framework for tackling various other optimization challenges that can benefit from genetic algorithm methodologies.

Fitness Evaluation Function

A critical component of our genetic algorithm is the fitness evaluation function, which assesses the quality of a candidate solution, i.e., a specific route through the cities. Each route is encoded as a permutation of the coordinates N [1, …, N], where N is the number of cities in the TSP instance. This encoding scheme represents the order in which the cities are visited.

The fitness evaluation function calculates the total distance of the route encoded by the permutation. To achieve this, we first calculate Euclidian distances between consecutive cities based on the sequence of city coordinates. Then, we sum the distances between these successive pairs of cities, including the return trip from the last city back to the first. This total distance serves as the route's fitness score, with shorter distances indicating better solutions.

This approach to fitness evaluation allows our genetic algorithm to effectively gauge the efficiency of routes, guiding the selection and evolution of candidate solutions towards the optimal or near-optimal route for the given TSP instance.

Extensive Testing and Parameter Optimization

To rigorously test our genetic algorithm and identify the optimal set of parameters for solving the TSP, we employed a Cross-Validation Grid Search technique. This systematic approach involves running the algorithm multiple times with different combinations of parameter values and evaluating the performance of each variant.

Genetic Algorithm Development

Genetic Algorithm Library Overview

This library is a modular and extensible implementation of a genetic algorithm (GA), aimed at solving optimization problems by simulating the process of natural selection. The core components of the library include classes for mutations, population management, crossover operations, problem definitions, selection mechanisms, and population generation. This structure facilitates the adaptation and application of the GA to various optimization problems beyond the TSP, showcasing the library's flexibility and power.

Core Components

- **Gene (gene):** A gene represents the basic unit of genetic information. In the library, a gene is defined as a class that encapsulates a specific value which could be a part of a solution to the optimization problem. The gene has attributes to hold its value and, depending on the problem type, boundaries that define the range of possible values it can take.
- **Chromosome:** A chromosome is a collection of genes. It represents a potential solution to the optimization problem. In the library, the Chromosome class holds a list of Gene objects.
- **Individual:** An individual is a distinct entity within the population that carries a chromosome. The Individual class encapsulates a Chromosome object, effectively wrapping the solution it represents.
- **Population (population):** A population is a collection of individuals. The Population class in the library manages a group of Individual objects, representing a diverse set of potential solutions to the optimization problem at any given generation. Manages groups of candidate solutions, tracking their fitness and enabling the selection of the fittest individuals for reproduction.

- **Mutations (mutations):** This module defines how genetic mutations are applied to individuals within the population. Mutations introduce variability, helping the algorithm to explore the solution space more thoroughly and avoid local optima.
- **Crossovers (crossovers):** Implements crossover operations that combine the genetic information of two parents to produce offspring. This is crucial for generating new candidate solutions that inherit characteristics from high-performing individuals.
- **Problems (problems):** Abstracts the optimization problem to be solved, allowing users to define the fitness function and any problem-specific constraints. This module makes the library applicable to a wide range of optimization tasks.
- **Selections (selections):** Encapsulates the selection strategy for choosing which individuals from the population will mate and produce offspring. This affects the balance between exploration of the solution space and exploitation of the best solutions found so far.
- **Population Generator (population_generator):** Provides mechanisms for generating the initial population of solutions, either through random initialization or more sophisticated methods, tailored to the problem at hand.

Key Features

- **Modular Design:** The library is structured in a modular fashion, allowing users to easily customize and extend its components. This includes different selection mechanisms, crossover and mutation operators, and fitness evaluation functions tailored to specific problem requirements.
- **Ease of Use:** With a well-defined API and structured approach to defining problems and configuring the GA, users can quickly set up and run optimization experiments.
- **Versatility:** Designed to be problem-agnostic, the library can be applied to a broad spectrum of optimization challenges, from route planning and scheduling to resource allocation and beyond.
- **Performance Monitoring:** The library includes mechanisms for tracking the evolution of solutions over generations, facilitating analysis of the algorithm's performance and the fine-tuning of genetic operators.

In summary, this genetic algorithm library offers a robust and flexible toolkit for solving optimization problems through evolutionary computation. Its design encourages experimentation and can be adapted to a wide range of applications, making it a valuable resource for researchers and practitioners in fields such as operations research, logistics, and beyond.

Specific Use case Overview

Our genetic algorithm (GA) is structured to efficiently tackle the Traveling Salesman Problem (TSP), leveraging the principles of natural selection and genetics, developed in our genetic python library, to explore the vast search space of potential solutions. Here is an overview of its design components:

- **Representation of Individuals:** In our genetic algorithm, each individual (or chromosome) represents a potential solution to the TSP and is composed of genes, where each gene corresponds to a city in the sequence of the tour. The entire chromosome is a complete sequence (or permutation) of all the cities in the problem, indicating the specific order in which the cities are to be visited. This representation ensures that each individual is a valid route that starts at a city, visits each city exactly once, and then returns to the starting city, thus forming a closed loop. The genetic operations—selection, crossover, and mutation—are carefully designed to manipulate these genes in a manner that explores the search space of all possible city sequences, seeking to find the shortest possible route that satisfies the TSP constraints.

- **Initial Population:** The initial population is generated randomly, ensuring a diverse range of starting points for the exploration of the solution space. The size of the population is a tunable parameter, balanced to provide sufficient genetic diversity while maintaining computational efficiency.
- **Selection Method:** We utilized a roulette wheel selection method (RouletteSelection), where the probability of an individual being selected for reproduction is proportional to its fitness relative to the population. This method simulates a roulette wheel where each slice is sized according to an individual's fitness score. The higher the fitness, the larger the slice, and thus, the higher the chance of being selected. This approach encourages the selection of fitter individuals while still permitting a degree of genetic diversity by allowing less fit individuals a chance to be selected. This balance is crucial for exploring the solution space effectively, ensuring that the algorithm does not converge prematurely on suboptimal solutions.
- **Crossover Operator:** The crossover technique utilized is the Partially Mapped Crossover (PMX), designed specifically for dealing with permutation problems like the TSP. This technique ensures that offspring receive a mix of parent genes while maintaining a valid tour. The expected impact is a balanced exploration of the search space, allowing the algorithm to efficiently combine beneficial traits from different individuals.
- **Mutation Operator:** We implemented a swap mutation (SwapMutation), where two cities in the tour are randomly selected and their positions are swapped. This mutation method introduces small, random changes to the individuals, helping to maintain genetic diversity within the population. This operator plays a critical role in preventing premature convergence and ensuring the population does not become too homogeneous. By occasionally introducing new genetic configurations, it helps the algorithm escape local optima and explore new areas of the search space.
- **Termination Condition:** The algorithm terminates after a predetermined number of generations have been produced.

Cross-Validation Grid Search Technique

The Cross-Validation Grid Search technique is divided into two main stages:

1. **Parameter Space Definition:** We first defined a grid that outlines a range of possible values for each parameter of the genetic algorithm. This grid represents the parameter space over which the search is conducted. For example:

```
# Define parameter grid
        param_grid = {
            "mutation_rate": [0.1, 0.05],
            "population_size": [50,80],
            "generations": [15000],
        }
```

8. **Algorithm Evaluation**: For each combination of parameters in the grid, the genetic algorithm was executed 5 times to solve the TSP instance. To ensure the robustness of our findings, we applied cross-validation by dividing the problem instances into subsets and rotating these subsets in a training-testing cycle. This method helps in assessing the algorithm's performance across different scenarios and reduces the likelihood of overfitting to specific problem characteristics.

The performance of each parameter set was assessed based on the quality of the solutions produced, specifically the total distance of the shortest route found. The evaluation criterion was not only the minimum distance achieved but also the consistency of the results across different runs, indicating the algorithm's reliability and stability.

Optimization Outcome

The outcome of the Cross-Validation Grid Search provided us with invaluable insights into how different parameters affect the genetic algorithm's performance on the TSP. By analyzing the results, we were able to identify the most effective combination of population size, mutation rate, crossover rate, and selection strategy for our specific problem instances. This optimized set of parameters significantly enhanced the algorithm's ability to find near-optimal solutions consistently, thereby improving its overall efficiency and effectiveness in solving the Traveling Salesman Problem.

This comprehensive approach to testing and optimization ensures that our genetic algorithm is not only tailored to the characteristics of the TSP but also fine-tuned to deliver the best possible performance, making it a robust solution for this complex optimization challenge.

Results

Cross-Validation Grid Search results

In the process of optimizing the Travelling Salesman Problem (TSP), a grid search cross-validation method was employed to identify the best parameter combinations for achieving good solutions. This method involved varying the crossover rate, mutation rate, and population size across some defined combinations to evaluate their performance impact on solving the TSP. The parameters varied were:

- Crossover rate: 0.5 and 0.9
- Mutation rate: 0.05 and 0.1
- Population size: 50 and 80

For each of the eight possible parameter combinations, the algorithm was run five times during 15000 generations each, and both the average and standard deviation of the results were calculated to analyze the outcomes comprehensively. This approach ensured a robust assessment of each parameter set's effectiveness in navigating the solution space of the TSP.

The experimentation culminated in a graphical representation of the results, highlighting the performance of each parameter combination in terms of the solution quality and convergence rate to the global optimum. The analysis revealed that the best parameter set consisted of a 0.9 crossover rate, a 0.05 mutation rate, and a population size of 80.

Interestingly, the graph also illustrated that a lower mutation rate of 0.05 consistently yielded better results across different trials, indicating its significant relevance in enhancing the algorithm's efficiency. Conversely, while a higher mutation rate of 0.1 allowed the algorithm to converge more rapidly in the initial stages of optimization, as we observed in the historic graphs generated after each run, this early convergence did not necessarily translate into better end solutions. This observation suggests that implementing a dynamic mutation rate—starting higher to encourage exploration and decreasing over time to focus on exploitation—could further optimize performance by balancing the exploration of the solution space with the refinement of promising solutions.

Additionally, the results pointed to the benefits of a larger population size, as expected. A bigger population offers a broader genetic diversity, facilitating a more comprehensive exploration of the solution space. However, this advantage comes at a computational cost, as larger populations require more resources to maintain and evolve. This trade-off underscores the necessity of carefully considering the computational budget when deciding on the population size, as the increased resource consumption may not always justify the potential improvements in solution quality.

In summary, the grid search cross-validation process provided good insights into the genetic algorithm's parameter tuning for the TSP. The analysis not only identified a good combination of parameters for this specific problem instance but also highlighted the importance of mutation rate dynamics and population size in the algorithm's performance. These findings can guide future optimizations and adaptations of our genetic algorithm for solving the TSP and similar complex optimization problems.

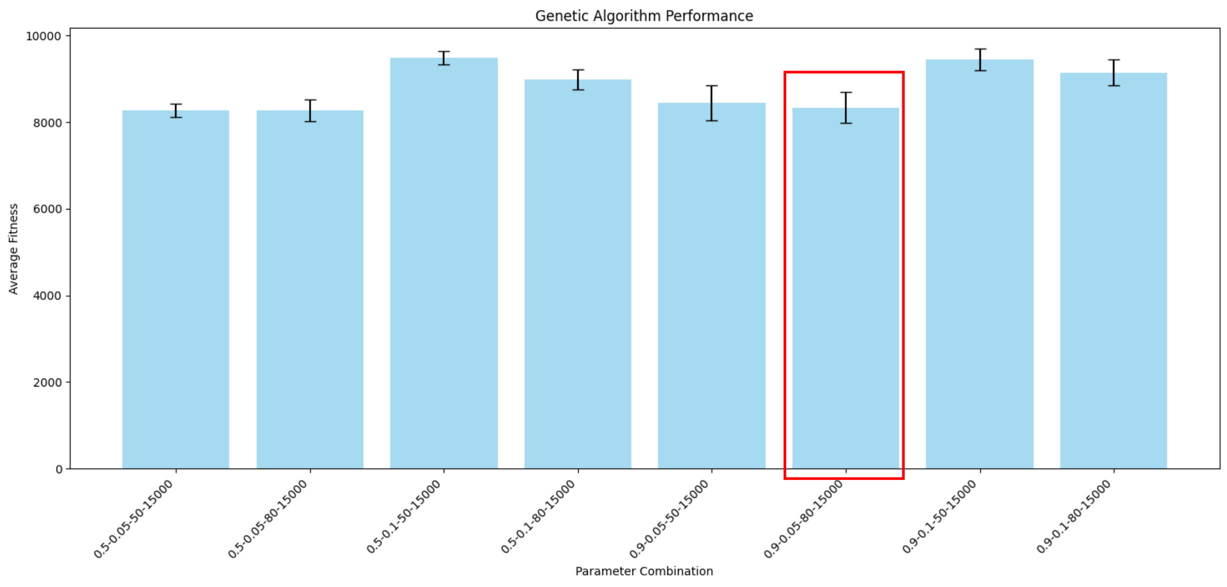


Figure 1 - Cross-validation Grid Search results.

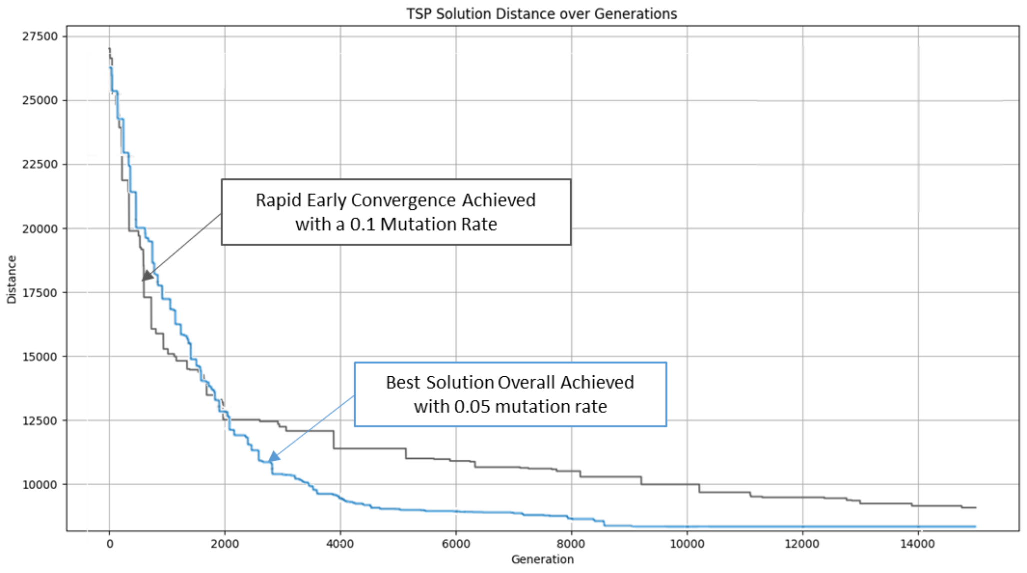
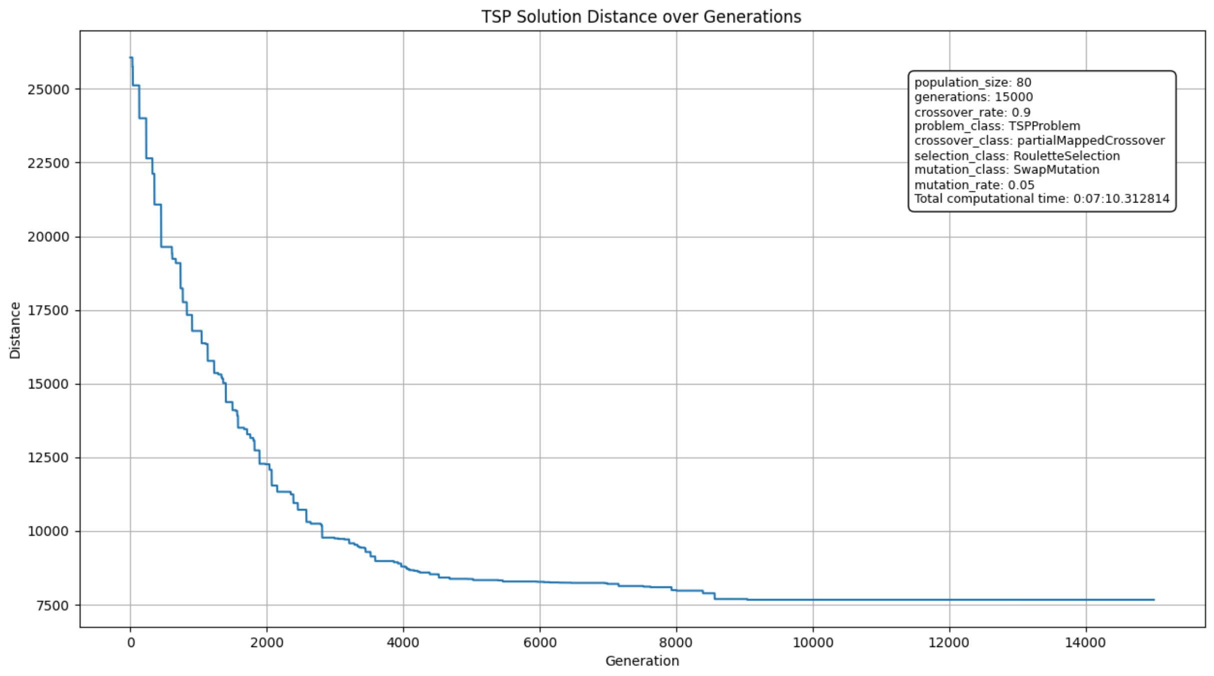


Figure 2 - Historic convergence comparation.

Solution Quality

Based on the algorithm's performance, the best route found for the Berlin52 problem from TSPLib has a total distance of approximately **7666** units. This is very close to the optimal solution listed in the TSPLib, which is 7542 units, with a error of less than **2%** (1.6%). This indicates that our algorithm was able to find a close to optimal solution for this problem instance, matching the best-known result.



population_size: 80
generations: 15000
crossover_rate: 0.9
problem_class: TSPProblem
crossover_class: partialMappedCrossover
selection_class: RouletteSelection
mutation_class: SwapMutation
mutation_rate: 0.05
Total computational time: 0:07:10.312814

Figure 3 - Best solution historic convergence curve.

Visual Inspection

The visual representations of the routes produced by the algorithm for the Berlin52 problem are provided in the plot. Theis plots demonstrates the paths taken between cities in the most efficient manner identified by the algorithm. The clear, concise paths without almost no unnecessary crossings indicate a well-optimized route.

Important: Although the plots don't show the path from the first city to the last one, the algorithm considered this distance in its fitness calculation.

Berlin52 Plot: The route visualization for Berlin52 shows a nearly optimal path connecting all cities without intersecting lines, which is characteristic of an efficient solution for the TSP.

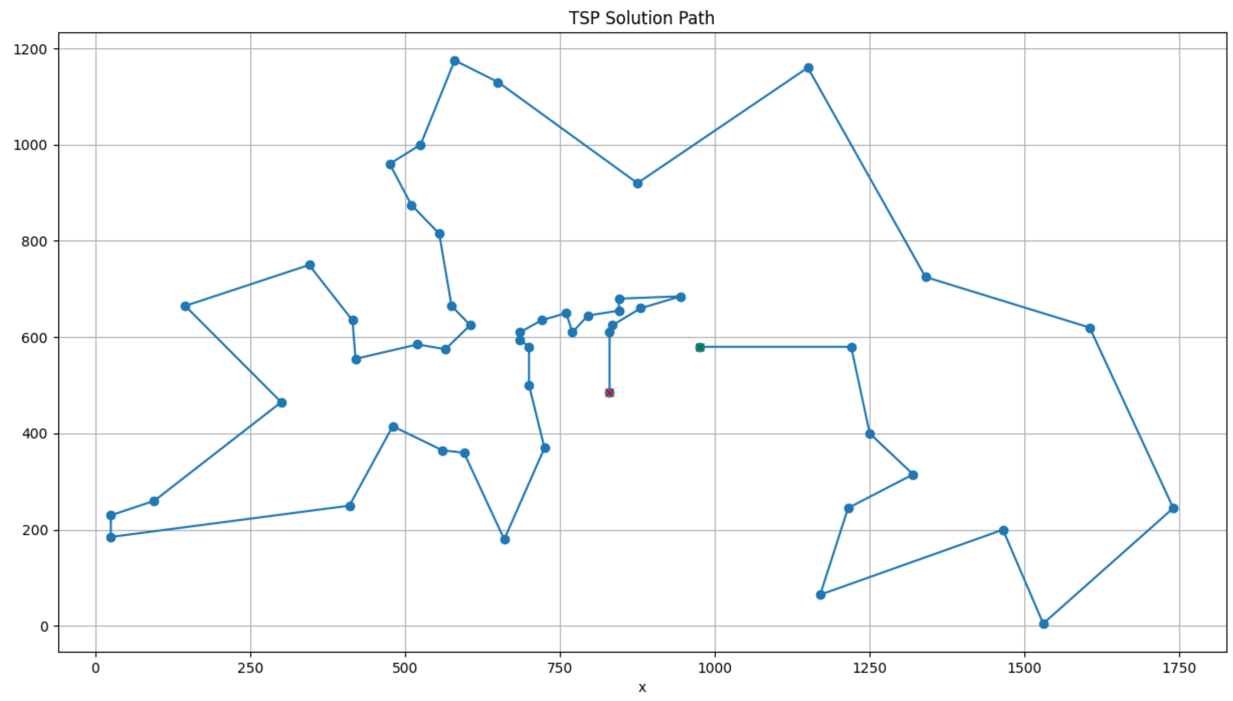


Figure 4 - Best solution path generated.

Conclusion

Evolutionary Operators Analysis

The evolutionary algorithm's success in approximating a solution close to the optimal path, particularly for the Berlin52 problem, underscores the efficacy of the selected evolutionary operators. The genetic operators—selection, crossover, and mutation—were developed to maintain a balance between exploration and exploitation of the search space. The selection process effectively prioritized high-fitness individuals, ensuring that promising solutions were retained for subsequent generations. Meanwhile, the crossover operator facilitated the combination of beneficial traits from parent solutions, promoting the emergence of superior offspring. The mutation operator introduced necessary diversity into the population, preventing premature convergence to suboptimal solutions.

Solution Evaluation

The algorithm achieved a solution with a total distance of 7666 units for the Berlin52 problem, which, while not matching the optimal solution of 7542 units from TSPLib, signifies a good performance given the problem's complexity. The slight deviation from the optimal solution underscores the algorithm's effectiveness in identifying efficient routes through the heuristic and evolutionary search processes. The visual representation of the route, characterized by its lack of intersecting lines, further attests to the solution's quality.

Challenges and Limitations

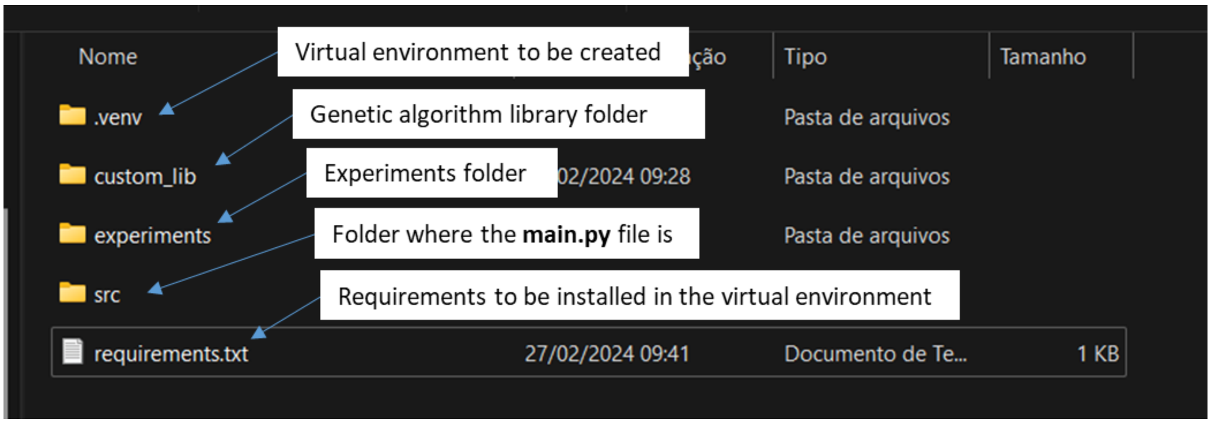
One of the main challenges faced during the project was the balance between exploration and exploitation. Ensuring that the algorithm could explore a wide range of solutions without getting trapped in local optima required fine-tuning of the evolutionary operators and parameters, like the mutation rate for example.

Additionally, the algorithm's performance is inherently stochastic, meaning that different runs can produce varying results. This variability underscores the importance of multiple runs and the use of statistical measures to assess the algorithm's reliability and performance accurately.

In conclusion, the project demonstrated the potential of evolutionary algorithms in solving complex optimization problems like the TSP. The results achieved for the Berlin52 problem illustrate the algorithm's capability to find near-optimal solutions. However, the challenges and limitations encountered also highlight areas for future improvement, particularly in enhancing computational efficiency and further refining the balance between exploration and exploitation to achieve even closer approximations of the optimal solutions.

Appendix

Working directory:



- **.venv:** Virtual environment folder that needs to be created in order to run the experiments and try out the algorithms created.
- **custom_lib:** Folder where the library created is stored to be installed in the virtual environment using pip and the requirements.txt file through:
    - *pip install -r requirements.txt*
- **experiments:** Where all the experiments data is stored after each run, here the graphs, parameters and final results can be analyzed.
- **src:** Where the code was created and where the main.py file is
- **requirements.txt:** The file to be used with pip to install the dependencies of the project