Karolína Rusnačková (525276), Jakub Čillík (524749)

Solver for the Travelling Salesman Problem using LNS

The solver is implemented in four files: `lns_solver.py`, `initial_solutions.py`, `destroy_methods.py` and `repair_methods.py`. Each file contains implementation of logic that the file's name suggests.

Solution Representation

The solution is represented as a **permutation** of positive numbers (a Python list) where each number represents a location index (indexed from 0). The order of numbers determines the route.

Infeasible Solutions

The permutation-based representation of the problem ensures that each solution is equal to the number of locations and that each location appears exactly once in the solution.

Initial Solutions

The first implemented method is a **random permutation**. Randomness showed potential at the start. However, since it requires a lot of iterations with small cost changes to get close to the desired solution, this approach was relegated. The second, more complex method is the **greedy approach**. Initially, a random city is selected and it iteratively appends the nearest one to the route.

Destroy Methods

We tried three different approaches: random removal, n worst cases and Shaw Removal.
Random removal selects the desired number of cities and removes them from the solution.
n **worst cases** removes n cities with the highest sums of the distance to their two neighbors.
Shaw removal calculates the sums of distances between the two neighbors for each city, then randomly selects a `seed city` and removes cities with similar distances to neighbors as the `seed city` has to its neighbors.
In the final implementation we use the random removal (see the Optimizer Section).

Repair Methods

We use the classical **greedy repair** in the solver. The method repairs a solution by greedily reinserting deleted locations at positions that minimize the overall solution cost. After the greedy repair, we try to improve the solution by calling the **2-opt** operator. It iteratively examines all possible pairs of cities in the solution to identify the best swap of 2 edges that results in the lowest cost. In case such a swap is found, the newly explored solution is updated.

Cost function

All the cost functions used in the solver use **incremental calculations**. Trivial calculation is also present but we used it only during development and testing.

Solution Acceptance

Simulated annealing is used to accept newly explored solutions. Compared to trivial accept, it helps to explore the solution space more thoroughly. The parameter `T_initial` used in the implementation as the initial temperature was chosen experimentally with manual tuning. The discount factor `alpha` was chosen as rather high to favor exploration in the case of many local optima. We apply exponential decay.

Termination Criteria

We decided to use the whole provided time-limit as termination condition. This seemed reasonable, as it allows the algorithm to explore the solution space as thoroughly as possible within the available time.

Parametrization

The solver takes a 2D matrix with location distances, a `Timeout`, and a file path where the solution will be present. Parameters for simulated annealing and methods are default parameters (discussed in the Solution Acceptance and Optimizer sections).

Generative AI Utilization

Karolína Rusnačková: I tried using ChatGPT for the implementation of k-opt. However, it did not work well after I tested it. I concluded it might be more efficient to write it from scratch following this pseudocode `https://en.wikipedia.org/wiki/2-opt#cite_ref-1`. Otherwise, I did not use AI.

Jakub Čillík: Similarly, I attempted to utilize the power of ChatGPT to provide a starting point for the required algorithms. The code provided was too generic to be used for this particular implementation. Therefore, I did not use anything from ChatGPT. However, I used ChatGPT as an assistant for the Python standard library documentation.

Author's Contributions

The workload division was reasonably balanced and we discussed the used approach together.

Karolína Rusnačková: I implemented the repair methods (greedy repair, 2-opt operator), LNS solver algorithm template with acceptance criteria (simulated annealing) and calculation of the cost (trivial and incremental) for all the methods.

Jakub Čillík: My task was mainly implementing the initial solutions (random, greedy) and destroy methods (random, n worst cases, and shaw removal). I also dedicated some time to the `debug.py` that focuses on testing and tuning our LNS solver. Lastly, I implemented an optimizer, which adjusts parameters and manages the use of particular methods actively on the run.

Optimizer

The `optimizer.py` was introduced for the purpose of analyzing and comparing the performance of the implemented methods. The logic behind is to try the default strategies and parameters iteratively. If no improvement appeared for a given number of steps, change the strategy and/or tweak the parameters.

Several observations have been made:

- Random initial solution creates a lot of space for rapid cost decreases in the beginning. However, the time spent improving it could rather be utilized better in the later iterations.
- Enabling 2-opt at any stage (even in the later iterations) proved that 2-opt is capable of making improvements even if other destroy and repair strategies are stuck.
- Using n worst cases helps initially by relocating cities with long distances, but stagnates and removes the same cities every iteration as the solution converges to (local) optima.
- In case of a lot of cities with the same distances to their neighbors, 2-opt and shaw removal showed a lot of potential. On the other side, n worst cases performed poorly.