Karolína Rusnačková (525276), Jakub Čillík (524749)

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 an implementation of logic that the file's name suggests. `lns_solver.py` contains the main loop and outsources work to the algorithms in dedicated files.

Solution Representation

The solution is represented as a **permutation of positive numbers**, internally as a Python list. Each number in the list refers to a particular city, and the order of the cities determines the route itself. E. g., `[1, 5, 3, 2, 4]` for 5 cities.

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** of the cities. 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 iteratively, it finds the nearest one, appending it to the route.

Destroy Methods

For destroy methods, the following algorithms are implemented: **random removal**, n **worst cases** and **shaw removal**.
Random removal, using non-determinism, selects the desired number of cities, removes them from the solution, and incrementally updates the cost.
n **worst cases** removes n cities with the highest sums of the distance to their two neighbors in descending order.
Shaw removal is the most complex of these three methods. It calculates the sums of distances between the two neighbors for each city. Then, it randomly selects a `seed city`, which will serve as a reference, and removes cities with similar distances to neighbors as the `seed city` has to its neighbors.
Various combinations and parameter tweaking were performed during testing and optimization. However, we settled on the random removal method. (See Optimizer)

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. The cost function is implemented incrementally, i.e., in constant time.
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. The calculation of the cost of the swap is also in constant time.

Cost function

The cost functions in the solver use **incremental calculations** to boost performance. The trivial calculation is also present, although used only during development and testing.

Solution Acceptance

Simulated annealing is utilized to accept newly explored solutions. Compared to trivial accept (always accept only improving solutions), this metaheuristic 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

Our TSP LNS solver relies on `Timeout` parameter provided in the instances. The decision to use such termination criteria is reasonable and allows to keep the solver running as long as possible to the given conditions. The main loop terminates as it detects it is over the timeout. Moreover, to prevent killing the solver process with no results at all, as some algorithms may take a little while to finish before the termination condition is checked again, **checkpoint saving** is implemented. Every time the solution significantly improves, it is saved to file as the best solution.

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 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 directly or indirectly. However, I used ChatGPT as an assistant for the Python standard library documentation. An example could be that instead of writing a *for loop*, I searched for an existing function in the Python standard library that provides the exact functionality using ChatGPT. This approach is much faster than reading documentation.

Author's Contributions

The workload of each participating author was reasonably balanced. Before the implementation, we met and discussed potential contributions and planned the approach. Throughout the entire sprint, we discussed our progress and the next tasks.

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

Jakub Čillík: My Task was mainly implementing the initial solutions and the destroy method. Namely, functions for initial solutions include random and greedy algorithms, whereas for destroy methods, these are random, destroy n worst, and shaw removal. I also dedicated some time to the `debug.py` that focuses on testing and tuning our LNS solver. Lastly, my contribution lies in the optimizer, which adjusts parameters and manages the use of particular methods actively on the run.

Appendix

Optimizer

Considering the various methods and algorithms implemented, a natural question arises: which one should be used and with which parameters? The **Optimizer** was introduced especially for the purpose of testing the methods, tuning the parameters, and optimizing the performance.
The logic behind the Opmizier is to try the default strategies and parameters iteratively. If no improvement has been made for a given number of steps, change the strategy (e.g., destroy method) and/or tweak the parameters for the given strategy.
Several observations have been made:

- Random initial solution initially creates a lot of space for improvements, giving rapid cost decreases. However, when the solver stagnates, there is little to no chance to improve further as the initial solution does not follow any rational procedure. Also, the time spent improving the random initial solution is expensive and could have been utilized better in the later iterations for fine-tuning.
- 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.
- n worst cases is usable at the beginning as there are a lot of cities in the incorrect places with long distances. This makes them perfect to be considered a 'bad case' following their removal and insertion in another place in the route. However, as the solution converges to optima, n worst case stagnates and removes the exact cities every iteration, beings of no use.
- Switching the destroy methods on the run improved the variance of the solution costs, narrowing too-good and too-bad solutions. However, for some topologies (grids), the performance was significantly worse than that of random removal.

The results showed a slight correlation between the methods used and the topology of the cities.

- When there are a lot of cities with the same distances to their neighbors, 2-opt and shaw removal (if it uses the appropriate city as a reference) showed a lot of potential. On the other side, n worst case performed poorly.
- If the cities are clustered around a point with up to $\frac{1}{4}$ outliers, the n worst case method showed potential as there is space to reorder the outliers ('bad case cities') differently, possibly improving the cost.

Testing

Each of the authors tested the algorithms they implemented using the Python `assert` command. The internal agreement was to write at least a few tests for the functions. However, the tests are absent in the submitted codebase as the authors used different IDEs and workflows, and the testing with argument passing would require effort outside the plan.