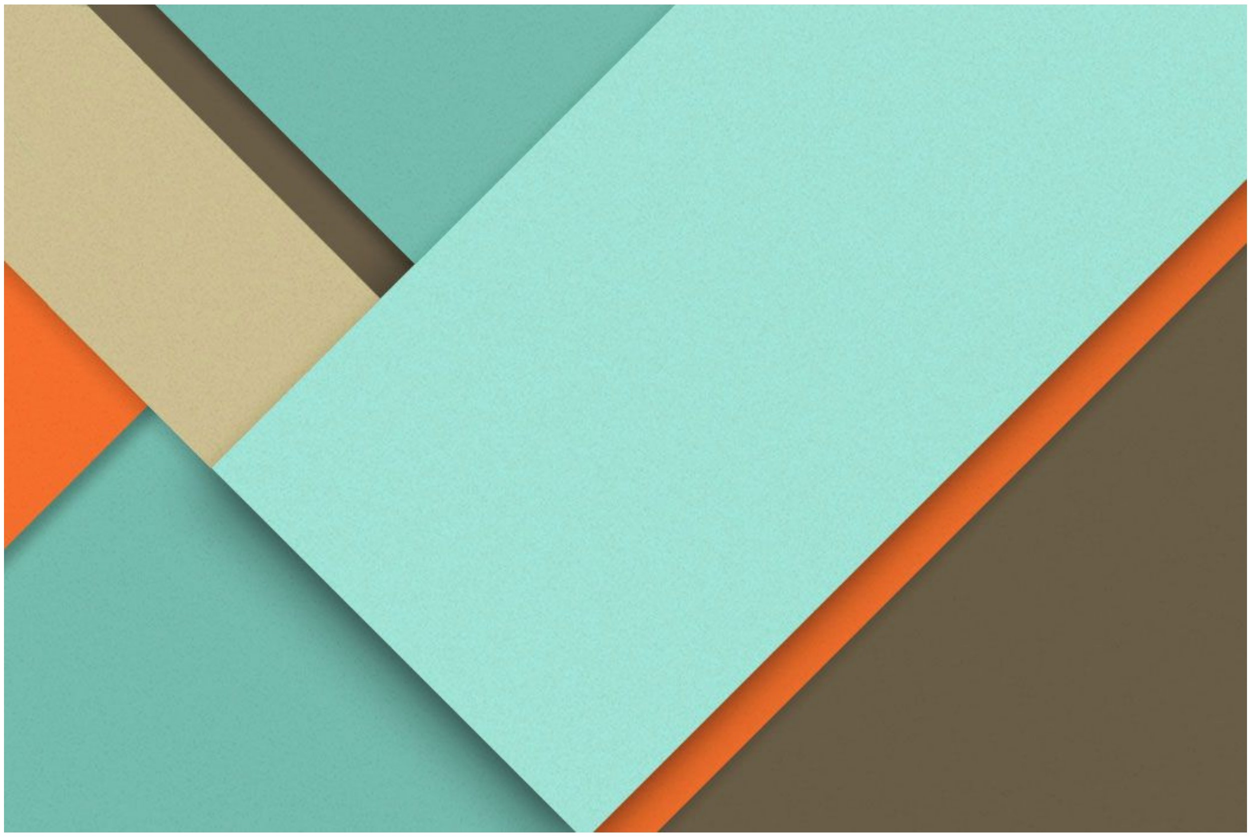

RO Project, Group l18

10.07.2018

Overview

We implemented Vehicle Routing Problem through the use of interactive Python notebooks; in particular, we implemented the VRP with Capacity and Linehaul and Backhaul routes. The code itself is only commented in the most obscure parts, since we tried to keep very long and “talking” variable names so that it was easier to navigate. The repository with the code is publicly available at this GitHub link:

https://github.com/sirbardo/ROProject

Specifications

The project is divided in three jupyter notebooks, one for the C&W sequential algorithm, one for the C&W parallel algorithm and one for the construction of the result tables.

Both the C&W sequential and parallel versions of the algorithm consist in a main loop that iterates over the instance files. For each instance file, an initial set of routes is computed, then a series of merges is performed on the initial routes based on the algorithm version.

After this phase (construction phase), the local search phase is performed by the iterative execution of the *find_best_relocate()* and *find_best_exchange()* functions. The loop ends when the difference between two successive objective function values is smaller than a given threshold.

To create the result tables it is necessary to run the CandW_Results after the execution of the CandW and CandW_Parallel notebooks. The tables will be saved on *sequential_results.txt* and *parallel_results.txt* files.

Code

Common functions:

- euclidean_distance2D(point1, point2): calculates the 2D Euclidean Distance between two points; the points are expressed as 2-tuples of numbers. Returns the distance as a number.
- is_route_valid (route_to_check): this is the function that takes a possible route as input and checks whether or not it fulfills all the constraints. Returns a boolean.
- is_route_merge_valid(id_route1, id_route2): checks if a merge between two routes can be done without breaking any constraint. It returns either an empty string if no merge is possible, the string ‘l’ if the left merge (route2 merged to the left of route1) is possible, ‘r’ for the right, and ‘lr’ for both.
- merge_two_routes(id_route1, id_route2, direction_of_merge): modifies the list of routes by merging the routes in id_route1 and id_route2, along the direction specified. For the sake of efficiency, we keep both former routes and set one of them to (-1, -1, -1) so that we do not need to resize our list at runtime.

- find_best_relocate(): tries all the possible relocates between routes and returns the one that doesn't break any constraint and has the most improvement to the cost function. It returns it as a tuple of (improvement_value, (id_route1, new_route_1), (id_route2, new_route_2)). This is done for the sake of efficiency, because it allows us to skip a few expensive copies and searches further down the algorithm.
- find_best_exchange(): finds the best (most improvement on the cost function) out of all the possible exchanges. Returns the same tuple as the previous function.
- find_best_improvement(): this is where the local searches are iteratively called until a given threshold of improvement is not satisfied anymore.
- get_cost_matrix(): calculates all distances between nodes and returns the matrix of costs from a node i to a node j.
- get_savings_matrix(): calculates the matrix of savings as specified in the specifications.
- compute_total_cost(): computes the objective function over the list of routes.
- merge_routes(): this is where the C&W algorithm is implemented. It differs because it implements the sequential version in the sequential notebook, and the parallel version in the other one.

Sequential only functions:

- find_first_merge(id_route1, available_merges): out of all the feasible merges, it picks the first one in the ordered savings list, as per specifications. This is used in the C&W algorithm pipeline.

Parallel only functions:

- there_is_a_possible_merge(i, j): this function is used as part of the C&W Parallel pipeline; it iterates over the routes until it finds the first possible merge (not necessarily feasible) between two routes, and returns it.

Results

Overall, we are satisfied of the results we obtained. The gaps between the optimal values given and our results are usually very small, and the code, considering it was written in interactive python, is efficient enough to run over all the instances in minutes.

In a case, we even managed to get a better result than the ones given. The detailed results can be found in the result tables, together with their computational times.