EINDHOVEN UNIVERSITY OF TECHNOLOGY

Built Environmnet

Assignment 1 report

Large Scale Optimization

Matteo Meloni (2389975)

Year 2025

Contents

1 Introduction

With the increase in demand for delivery services, as in e-commerce and food delivery, there is an increasing need for routing solutions. One of the most common problems is the Travelling Salesman Problem (TSP). The following lines show the problem definition:
A traveling salesman likes to visit n cities, starting from and ending at city 1. The distance between the cities i and j equals d_{ij}, for $1 \leq i, j \leq n$. We want to find the shortest route visiting all cities.
While it is fairly easy to find a feasible solution (any tour visiting all cities is feasible), it is much more challenging to find the optimal solution. In fact, given a problem with n cities, there are $\frac{(n-1)!}{2}$ possible tours (we are dividing by 2 since each tour can be travelled in two directions). It is clear that comparing every possible tour is not a viable option, hence the need for more efficient algorithms.
In this report, we will explore some methods to solve the TSP, mainly the Greedy Randomised Adaptive Search Procedure (GRASP).

2 Methodology

2.1 Selecting files

In order to develop and test the algorithms, several files were selected from the provided instances. Specifically, 10 instances were chosen at random according to the following criteria:

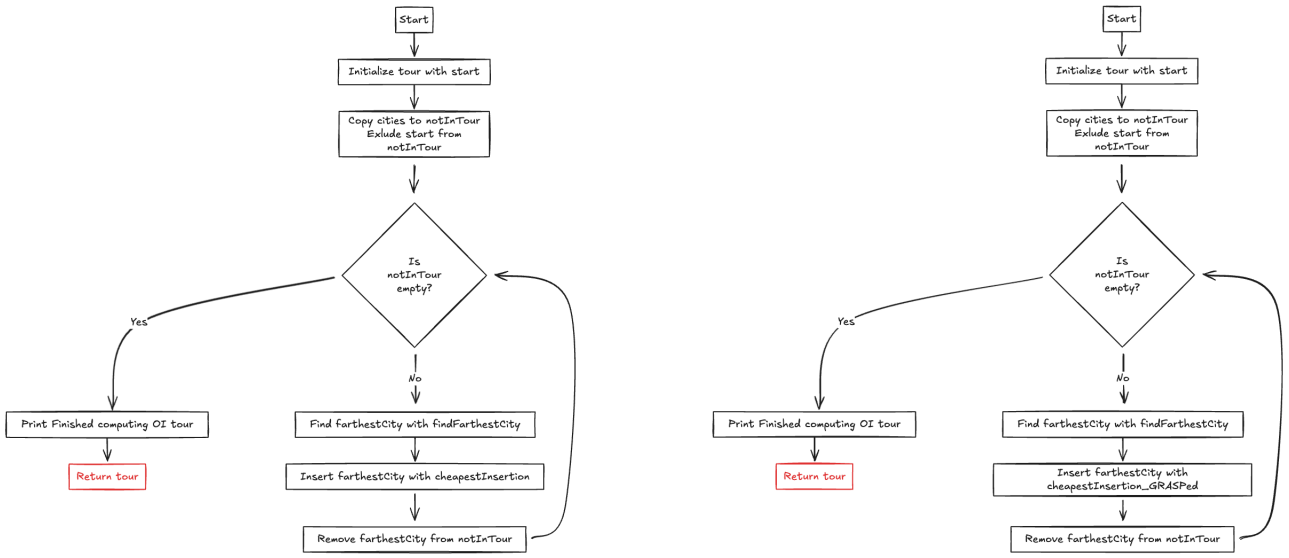
-	Small	Medium	Large
1	kroA150.tsp	ali535.tsp	pla7397.tsp
2	pr107.tsp	rat195.tsp	pr2392.tsp
3	eil51.tsp	rat575.tsp	-
4	ch130.tsp	-	-
5	ulysses22.tsp	-	-

Table 1: Selected instances

Note, that the number in the name of each instance indicates how many cities it contains.
On top of these, some extra test instances were added as needed. These are very simple, manyally built instances, with only few cities with simple coordinates, designed to test specific functions in the code. The TSP for these instances can be solved manually, or, at very least, they can give an intuition of the solution. This is useful to check that the code gives the expected results, before applying it to the regular instances. These instances are: *smallTest1.tsp*, *point5_test.tsp* and *point5_test2.tsp* and are attached to the report with the regular instances.
Code was developed in Python. Instances get initialised into a TSP object, which calculates a distance matrix upon initialisation. The TSP class also contains methods to solve the problem applying different algorithms. The main algorithms implemented are (with the methods' name in the code):

- Nearest Neighbour (NN) - *getTour_NN*
- Outlier Insertion (OI) - *getTour_OutlierInsertion*
- GRASPed Outlier Insertion (GRASPed OI) - *getTour_GRASPedInsertion*
- 2-opt local search - *makeTwoOpt*

The report will not go into the details of the NN, as it was already implemented in the provided code.



(a) Outlier Insertion flow diagram

(b) OI with GRASPed insertion flow diagram

Figure 1: OI flow diagrams (with and without GRASP)

2.2 Outlier Insertion

The first method tested was the Outlier insertion. Its flow diagram can be seen in Figure 1a. The *cheapestInsertion()* method inserts the farthest city in the tour, in the position that minimizes the increase in cost. To find this position, the method loops through all arcs in the existing tour, adds the new city on the given arc, and calculates the cost increase of this operation. Whenever it finds a better cost increase, it saves the position. At the end of the loop, the city is inserted in the best position found.

Let the cost increase be Δ, c_{ab} the cost of arc (a, b) and i the position being tested. The cost of insertion is calculated as:

$$\Delta = c_{i-1,i} + c_{i,i+1} - c_{i-1,i+1} \tag{1}$$

Figure 1 show the flow diagrams of the OI and GRASPed OI algorithms. Clearly, the only difference stands int

2.3 GRASPed Outlier Insertion

The second method introduced is the GRASPed Outlier Insertion. There are 2 steps that can be randomized in the OI algorithm:

- *findFarthestCity()*: instead of taking the farthest city, we could create a Restricted Candidate List (RCL) of the n farthest cities, and pick one at random
- *cheapestInsertion()*: instead of inserting the city at the location that would cause the lowest increase in tour length, we could find a RCL of insertion points and select at random inside of it where to insert the city in the tour

In this case, only the latter was implemented. *cheapestInsertion_GRASPed* performs the same operations as *cheapestInsertion*, but instead of saving only the best position, it saves the best n

positions. Then, it selectes one at random and inserts the city there. The size of the RCL can be set as a parameter when calling the method.

2.4 2-exchange

Finally, the 2-opt local search was implemented. The method *makeTwoOpt* takes a tour as an input and check if it is two-optimal. To do so, the method *isTwoOpt* is called. *isTwoOpt* loops through all pairs of arcs in the tour and computes the cost variation (saving or Δ) of swapping them (as in Eq 2). As soon as it finds a pari that, if swapped, would reduce the tour cost, it returns "False" and the indices of the cities to swap. If no pair is found, then it return "True". *makeTwoOpt* then takes the 2 indices and recunstructs the tour performing the 2-opt swap. The process is repeated until the tour is two-optimal.

$$\Delta = -c_{i,i+1} - c_{j,j+1} + c_{i,j} + c_{i+1,j+1} \tag{2}$$

2.5 Running for multiple starting points

Every algorithm is run through the *runForMultipleStartingPoints* method. This method takes three inputs: a file path to a .tsp file, used to create TSP object; the number of starting points to use; the algorithm to use. After running the chosen algorithms, it stores the results into a dataframe that is both returned as a variable and saved in a directory as a csv. The information stored in the dataframe includes the file name, the starting city, the cost before local search (if it applies, otherwise 0), the final cost, the computation time, the size of the restricted candidate list (if it applies) and the final tour.

3 Results

3.1 Changing starting point

The first experiment was to run the nearest neighbour algorithm multiple times, changing the starting point each time. The idea behind this is that the NN is a greedy algorithm, and it is very sensitive to the starting point. Therefore, running it multiple times with different starting points should give different results, and the best result can be selected.
For example, a run done on instance "pr2392.tsp" with 100 starting points chosen at random, gave a range of costs from 488740 to 461062, with a variability around the mean of 5.8%.
On the contrary, the same experiment on instance "kroA150.tsp" (a small instance) with 30 starting points chosen at random, gave a range of costs from 35377 to 32021, with a variability around the mean of only 10.0%, suggesting that changing the starting point has a smaller effect on bigger instances than it has on smaller ones. Nevertheless, there is a chance that this is due to the fact that a smaller percentage of starting points compared to total number of cities was used on the bigger instance.
It took almost 30 seconds to run 100 starting points on "pr2392.tsp". Although results were not saved, it took more than 4 minutes to run on "pla7397.tsp", which is starting to be a significant amount of time and quite impractical, suggesting that there is room for improvement in the algorithm.

3.2 Outlier Insertion

A medium instance, "ali535.tsp", was used to compare the NN and OI algorithms.

OI was able to find a better solution than NN, with a cost of 2214.3 compared to 2452.9 of NN (comparing the best solution for each method, in a pool of 100 different ones obtained changing the starting point). This though, came at a significant cost in computation time: while NN only took 1.45 seconds, OI took 260.5 seconds, almost 4 minutes. which seems not worth it considering an improvement of only 9.7% in cost. On the other hand, the maximum cost found by OI (2278.0) is still lower than the minimum cost found by NN (2452.9), suggesting that OI can consistenlty provide a better solution than NN.

3.3 GRASPed Outlier Insertion

Once established that OI can provide better solutions than NN, independently from the starting point, the next step is to see if adding randomization in the OI method can provide even better solutions. In order to do so, both OI and GRASPed OI were run again on medium instances "rat195.tsp" and "rat575.tsp" for different values of RCL sizes: 2, 3, 4, 5, 10, 15, and 1, that is equivalent to running the Outlier Insertion method. Tables 2a and 2b shows the outcomes of this method. This method was run only for one starting point.

RCL size	OI cost	GRASPed OI cost
2	2984.0	**2857.5**
3	2984.0	3078.8
4	2984.0	2887.7
5	2984.0	2962.4
10	2984.0	3030.3
15	2984.0	2973.8

(a) Tour cost for runs at different RCL size ("rat195.tsp").

RCL size	OI cost	GRASPed OI cost
2	8243.4	8638.6
3	8243.4	**8283.5**
4	8243.4	8676.6
5	8243.4	8689.8
10	8243.4	8465.1
15	8243.4	8544.7

(b) Tour cost for runs at different RCL size ("rat575.tsp").

Figure 2: Results for GRASPed OI

In both cases, best results were obtained for smaller RCL size of 2 or 3, even though OI without randomization turned out to consistently be the method leading to the best cost.

3.4 GRASPed Outlier Insertion with local search (2-opt)

To evaluate the 2-opt algorithm, GRASPed OI + 2-opt local search was run. Costs for 30 iterations (changing starting point each time) were stored both before and after running makeTwoOpt. Figure 3 shows the two costs for each run.
It is worth noting how the points live generally on a line but it's clear the importance of applying local search to reduce cost. Also, the best overall cost after local search (in green) was not given for the same run as the best overall cost without running a local search algorithm (in red).

3.5 Efficiency improvements

The reader might have noticed that large instances were never mentioned. At this stage, the code is quite inefficient and computation times add up quickly when more than just a few hundreds cities are involved. This is an extreme limitation, as one of the main benefits of heuristics is to limit computation time compared to finding exact solutions to the problem (assuming there exist a method to find the optimal solution).

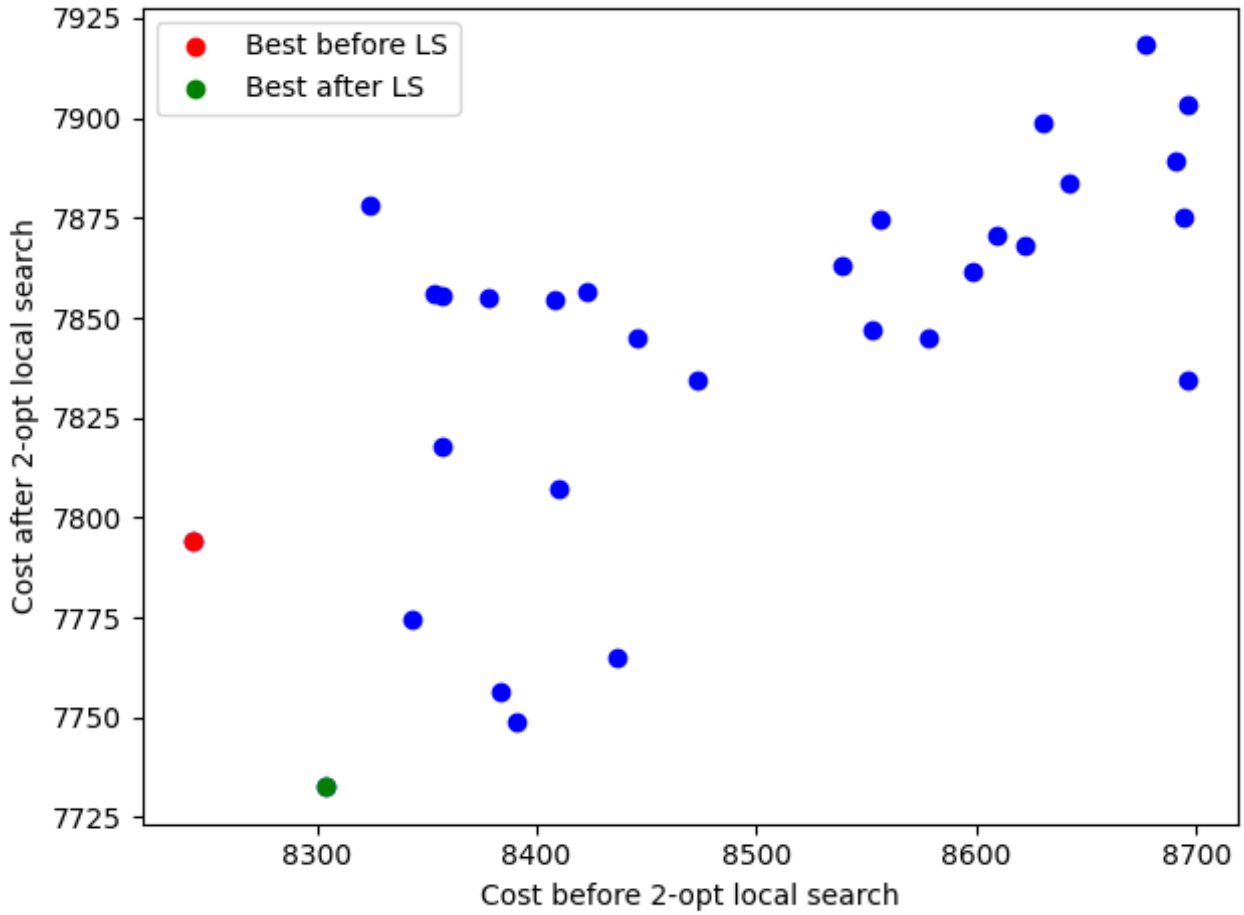


Figure 3: GRASPed OI + 2-opt results for 30 different runs

The first improvement made was in the *isTwoOpt* method. Initially, the method would always look from the first city in the tour every possible pair of arcs until it found one that was not 2-opt. As the tour got untangled though, the algorithm would still have to check every city from the beginning where the likelihood of finding a non-optimal pair of arcs is very low. Now, the method stores where the last pair was found, and only start looking for new pairs from that point. Only if nothing is found, does it run again through the whole tour to ensure that it is, in fact, 2-Opt. The example from §3.4 was run both with and without the improvement and the total computation time (for 30 different starting points) went down from 265 seconds to 88.

Another big improvement came from the realization that sometimes, the delta calculated by *isTwoOpt* would return $-1 * 10^{-13}$. In practice, this is equal to zero and it's not worth exchanging cities. This is probably the result of some internal approximation error. To solve this, the methods returns "not 2-opt" only if delta is less than 1. Before this, the whole program was unusable for large instances.

Just with these 2 fixes, the large instance "pla7397.tsp" was solved applying NN and 2.opt in about 22 seconds.

Still, there is a lot of room for improvement, even in other methods.

4 Conclusions

This document, explored several methods of solving the Traveler Salesman Problem. A number of algorithms were built, from constructive heuristics like outlier insertion (§2.2) to local search methods

like 2-opt exchange (§2.4). Overall, the experiments run seem to confirm that local search improves the results of a constructive heuristics and is therefore worth applying. On the other end, randomization in OI doesn't seem to improve the final outcome.

Despite the effort, attempts at improving the code substantially failed, and the current algorithms, at least for the way they are implemented, stay quite inefficient. The code struggles with dealing with large instances, even though the updates introduced in §3.5 allow to solve them for a limited number of starting points.

A Use of external resources

In the coding process, a number of external resources were used. In this appendix, the type of use will be described. These include ChatGPT, Copilot, Stack Overflow and modules documentation.
The main principle that drove the use of these resources was to "not ask about the logic or flow of an algorithm". I always developed the algorithms indipendently. On the other hand, given this was my first time using Python, and my limited knowledge of programming overall, AI was used mainly as a shortcut to python's documentation.
Prompt examples:

- How do I define a list in python?
- How do I get the last element of a list?
- What's the best module to create a plot?
- How can I save a data frame to a csv?

Examples of what was **not** asked or even looked up on the internet:

- How can I write the Outlier Insertion algorithm in python?
- Can you write a function to test whether a tour is two optimal?

There was one isntance where I had ChatGPT check why *getTour_OutlierInsertion* was taking so long to run, and it recommended to not compute and compare the cost of the full tour but rather just of the insertion. I took on this recommendation and found implemented it myself.