Built Environmnet

Solving Rich Vehicle Routing Problem using Adaptive Large Neighborhood Search

Large Scale Optimization

Mehrshad Ghorbanisharif (20731137)
Matteo Meloni (2389975)

Year 2025

Contents

1 Introduction

In this document we develop a programme to solve the Rich Vehicle Routing Problem (RVRP) with time windows. This problems consists in delivering goods with trucks that start and end the trip from a depot. In each route, the trucks pickup goods at a location and have to deliver them at a different location. A pair of pickup location and delivery location is called a request. An extra layer of complexity is added by the fact that each location only has a limited time window in which it can be visited. We solve the problem using a Adaptive Large Neighborhood Search (ALNS).
ALNS starts from an existing solution and modifies it trying to get closer to an optimal solution. The following section (§2) discusses how we build our algorithm. The general idea is to run several iterations; at each iteration we remove part of the requests from the solution using a destroy operator and we reinsert them using a repair operator. The amount of requests that we remove is selected at random at each iteration in a predefined range. At each iteration, we check wether the new solution is a new global best, a better solution than the previous one or a worse solution and based on this we reward the operators. At the following iteration, if an operator performed well will be more likely to be selected. This is what makes the algorithm *adaptive*.
To avoid getting stuck in local minima, we make use of simulated annealing (see §2.3): we sometimes accept a worse solution. On top of these, we tried improving our program both implementing new features, like electric vehicles, and applying 2-opt local search.
Section 2 describes all the methods used, both to build the operators and to tune corresponding parameters. Section 3 discusses the effectiveness of our program.
Our codes is quite effective at solving the problem. Our operators are able to consistently produce better results than the initial random operators. 2-Opt is also able to sometimes improve the solution. No single operator stands out of the others. Simulated annealing is sometimes able to provide better results, especially in the larger instances, suggesting that without it, the problem gets stuck in local minima.

2 Methodology

This section presents and explain the functioning of our program. The first part discusses the operators used; the second and the third part explain respectively the functioning of the adaptive functionality and how we included simulated annealing. The fourth section illustrates how each parameter in the program is calibrated. Finally, we discuss some general improvements, as picking a different initial constructor, including electric vehicles and applying 2-Opt local search.

2.1 Operators

We can count a total of seven operators: 4 destroy operators (Random removal (§2.1.1), Shaw removal (§2.1.2), worst removal (§2.1.3) and route removal (§2.1.4)) and 3 repair operators (Random insertion (§2.1.5), greedy insertion (§2.1.6) and regret insertion (§2.1.7)). They each use a different criterion to remove requests from an existing solution and then reinsert them, building a different solution.

2.1.1 Random removal

The Random Removal operator is the simplest of all destruction heuristics, and its main purpose is diversification. It works by randomly selecting a specified number of requests (n) from the current solution and removing them (Ropke & Pisinger (2006)).

2.1.2 Shaw Removal

We include a simplified version of the shaw removal algorithm used by Ropke & Pisinger (2006). The shaw removal operator takes several factors to define a relatedness (R) parameter. This parameter is a measure of how similar two requests are. The Shaw destroy operator works under the assumption that if we remove from a solution requests that are similar, it will be easier later to reinsert them using a repair operator. Unlike Ropke & Pisinger (2006), we only use two factors to define R: distance and demand.

$$R(i,j) = \varphi \left(d_{A(i),A(j)} + d_{B(i),B(j)} \right) + (1-\varphi) |l_i - l_j| \tag{1}$$

In equation 1, d and l are distance and demand. $d_{A(i),A(j)}$ is the distance between pickup location of request i to pickup location for request j, while B represents the delivery location.
This operator sorts all requests in order of relatedness to a random request and removes the n requests with greater R.
φ is a parameter that weights the importance of distance or demand in calculating the relatedness. The higher is φ, the more the relatedness is more related to distance than demand.

2.1.3 Worst Removal

We used the *Worst Removal* operator introduced by Ropke & Pisinger (2006). This operator evaluates how much the objective function improves when a given request i is removed from the solution.
The cost of removing a request is defined as:

$$\text{Cost}(i,s) = f(s) - f_{-i}(s) \tag{2}$$

where $f(s)$ represents the total cost of the current solution, and $f_{-i}(s)$ is the cost of the solution after completely removing request i. A higher value of $\text{Cost}(i,s)$ indicates that request i is poorly placed within the solution and that its removal provides a larger benefit.
Once the cost values for all removable requests are computed, they are sorted in decreasing order of $\text{Cost}(i,s)$, so that the requests offering the greatest improvement upon removal are considered first.
All requests are then sorted in descending order of cost, meaning that requests with the highest $\text{Cost}(i,s)$ values are considered first for removal. To avoid removing the same requests repeatedly, a controlled randomization is applied. The index of the request to remove is determined using the following equation:

$$\text{Index} = \lfloor |L| \times (\text{randomN})^p \rfloor \tag{3}$$

In equation 3, $|L|$ is the number of removable requests, randomN is a random number in $[0,1)$, and $p \geq 1$ is a parameter that controls the level of randomization. A higher p value makes the removal more deterministic, favoring the requests with the highest cost, while a lower p introduces more randomness in the selection.

2.1.4 Route removal

The route removal operator is a conceptually simple algorithm that consists in removing all requests from a selected number of routes. We allow removing from one to 75% of the total number of existing routes (Demir et al., 2012).

2.1.5 Random insertion

The Random Insertion operator works by reinserting unserved requests into the current solution at randomly selected feasible positions. It selects both the request and the route randomly, ensuring that every insertion opportunity has an equal chance of being chosen.

2.1.6 Greedy insertion

The greedy insertion operator always performs the best possible insertion. For each request not served, it finds the route and the location in that route that has the lowest insertion cost. It checks every request and starts inserting the one with the minimum overall insertion cost (Ropke & Pisinger, 2006).

2.1.7 Regret Insertion

The Regret Insertion operator, specifically implemented as the *Regret-k Heuristic*, is a powerful repair operator that aims to overcome the myopia of the basic Greedy Insertion. It achieves this by introducing a look-ahead mechanism that focuses on requests that would suffer the largest cost penalty if they could not take their best available position. The design of this operator is based on the framework presented by Ropke & Pisinger (2006) .
The request chosen for insertion is the one that maximizes the following expression, where U is the set of unserved requests, $\Delta f_{i,x_{ij}}$ is the insertion cost of request i in its j-th best route x_{ij}, and $\Delta f_{i,x_{i1}}$ is the insertion cost in its best route:

$$i^* = \arg\max_{i \in U} \left\{ \sum_{j=2}^{k} \left(\Delta f_{i,x_{ij}} - \Delta f_{i,x_{i1}} \right) \right\} \tag{4}$$

This term measures the regret value as the difference between the best and alternative insertion costs. We calculate the *regret value* by summing the differences between the best insertion cost and the next $k-1$ best alternative costs. The request i^* that maximizes this regret value is selected and then inserted into its minimum-cost position (the best route). Ties in the maximum regret value are broken by choosing the request with the lowest absolute insertion cost ($\Delta f_{i^*,x_{i1}}$).
The parameter k is the core of the Regret Insertion operator, controlling the extent of the look-ahead mechanism in what is officially termed the Regret-k heuristic. This controls the number of the best possible insertion opportunities that are examined in order to compute the overall regret. Larger k specifically implies that the algorithm examination of insertion costs on the request's k best routes enables it to compare the best single choice with a wider range of possibilities.

2.2 Adaptive

In order to make our program adaptive, we added weights to the operators. At the beginning, each operator has the same weight: 5. This means that, at the first iteration, each operator has the same probability of being selected. After each iteration, based on whether the new solution is accepted or rejected, the weights of the operators used get updated. We account for the four following possible scenarios: the new solution is the best solution found so far, the new solution is better than the previous solution, the new solution is worse than the previous one but accepted, the new solution is rejected. Table 1 shows the score for each scenario.

Score Ψ	**Description**
10	Best global solution
8	Better solution
5	Accepted solution
1	Rejected solution

Table 1: Score adjustment parameters

Let ρ_i be the weight of operator i, Ψ the reward according to table 1 and λ a decay parameter that controls how sensitive the weights are to changes. Then, weights are updated according to the following formula:

$$\rho_i = \lambda\rho_i + (1-\lambda)\Psi$$

The decay parameter is calibrated following the procedure in §2.4.

2.3 Simulated Annealing

To improve the exploration capability of the algorithm we integrated a Simulated Annealing (SA) acceptance criterion into the ALNS framework. The SA mechanism allows the acceptance of worse solutions with a certain probability, enabling the search to escape local optima and explore new regions of the solution space.
The acceptance probability of a new solution s' compared to the current solution s is defined as:

$$P(\text{accept}) = \begin{cases} 1, & \text{if } f(s') \leq f(s) \\ \exp\left(-\dfrac{f(s') - f(s)}{T}\right), & \text{if } f(s') > f(s) \end{cases} \tag{5}$$

where $f(s)$ and $f(s')$ represent the total cost of the current and new solutions, respectively, and T is the temperature parameter controlling the acceptance probability of inferior solutions.
Instead of manually setting the initial temperature, T_{start} is calculated dynamically based on the quality of the initial solution. This ensures that the annealing process is appropriately scaled for the specific instance size and cost range.
The starting temperature is chosen such that a solution that is $w\%$ worse than the initial solution has a 50% probability of being accepted. This can be expressed mathematically as:

$$P(\text{accept}) = \exp\left(-\frac{f(s') - f(s)}{T_{\text{start}}}\right) = 0.5 \tag{6}$$

where $f(s') = f(s) \times (1 + w/100)$. Solving for T_{start} gives:

$$T_{\text{start}} = \frac{f(s') - f(s)}{\ln 2} = \frac{f(s) \cdot w/100}{\ln 2}. \tag{7}$$

The temperature T is reduced at every iteration by multiplying it by a constant factor c $(0 < c < 1)$, which is the cooling rate parameter. As T decreases, the probability of accepting a worse solution drops, making the search behave more like a deterministic local search toward the end of the process.

$$T_{n+1} = T_n \times c \tag{8}$$

Both the start temperature control parameter w and the cooling rate c are tuned during the parameter tuning phase. The Simulated Annealing implementation in this report is based on the work of Ropke & Pisinger (2006).

2.4 Tuning

To establish an efficient baseline for the parameter calibration experiments, we first analyzed the convergence behavior of the ALNS algorithm. Using the default parameter settings, the algorithm was run on each instance to generate an iteration-versus-best-cost plot. Visual analysis of these profiles, such as the examples shown for lr205 and lrc104 (figure 1), indicates that the rate of improvement diminishes significantly after an initial period. Based on this observation, we establish an initial limit of 50 iterations for the tuning phase, dedicating the full iteration budget to finding the optimal parameters. After tuning all parameters, another experiment is conducted to study the effect of the number of iterations.

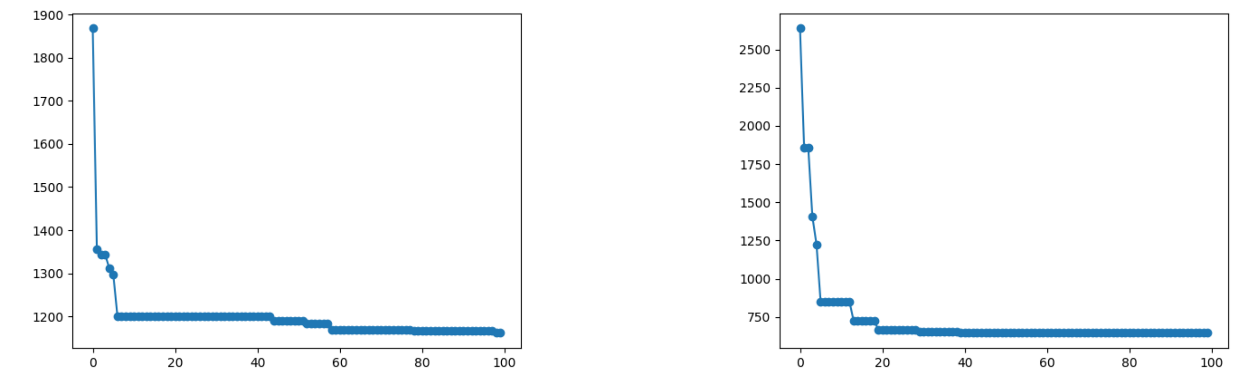

Figure 1: Convergence behavior for instance exploration, comparing two different problem types lc108(right) and lrc104(left). The maximum iteration limit was set at 50.

For tuning the parameters, we first focus on the local parameters of each operator: p for the Worst Removal, k for the Regret Insertion, and α and β for the Shaw Removal. The algorithm is run for each instance using three different random seeds to have broader exploration. We then calculate the average of the best solution costs obtained for each parameter setting. During this process, only one parameter is varied at a time while keeping all others constant. Table 2 shows an example of the parameter tuning results for the Worst Removal operator.

p	**Average Best Cost**
5	756.40
10	759.50
3	762.04
1	772.12

Table 2: Example of parameter tuning results for the Worst Removal operator.

After tuning the local parameters, we proceed to tune the global variables, such as the minimum size of the neighbourhood, the maximum size of the neighbourhood, and the update speed for the Adaptive mechanism, as well as the cooling rate and the initial temperature control for the Simulated Annealing process. In the case of Simulated Annealing, the cooling rate is tuned first, followed by the initial temperature control parameter. After tuning all parameters, we once again adjust the number of iterations. For each instance, we identify the iteration at which the solution cost decreases by less than 1%. The results of all tuned variables are presented in §3.1.

2.5 More improvements

This section discusses several improvements and adds-on to the program. These include choosing a different operator to construct the initial solution, making each route 2-opt and adding electric vehicles.

2.5.1 Initial constructor

Our program includes three different insertion operators: random insertion (§2.1.5), greedy insertion (§2.1.6) and regret insertion (§2.1.7). We can check, for each instance, which operator is best suited to build an initial solution. In order to do so, we run the program for each instance with each of the repair operators as initial constructor. We then compare the average cost across all instances to see which one leads to better overall results. Results are discussed in §3.

2.5.2 Electric vehicles

An extra layer of complexity in the RVRP can be added using Electric Vehicles (EV) instead of normal trucks. EVs in some routes don't have enough charge to complete the route itself. Our implementation stops at including battery charge that is used thought out the route. For each location, we calculate how much battery has been used and substract it to the total capacity. If there is not enough charge, the route is considered unfeasible and a new route is built. In some instances, there are routes that, even if they only contain one request, cannot be completed in one charge, making the request unserviceable.

2.5.3 2-Opt Local Search

The 2-Opt is a classical local search method that is used to improve routes by eliminating crossovers and reducing overall tour cost. It works by checking if swapping two edges in a route can improve the solution.

In our implementation, we apply 2-Opt to each route to improve its quality. The 2-Opt procedure was tested at different stages of the algorithm: after all iterations are completed, during each iteration following the application of the ALNS operators, and only when a new global best solution is found. The performance of these strategies is further discussed in §3.6.

It is important to note that even after applying 2-Opt, a route may not be fully 2-optimal. Constraints such as time windows, pickup and delivery requirements, and vehicle capacity can make certain 2-Opt moves infeasible.

3 Results

The following section presents the results of the conducted experiments. It begins with the parameter tuning outcomes, followed by a comparison between the final solutions obtained with the tuned parameters and the initial solutions. Subsequently, the impact of several algorithmic modifications is examined, including the choice of initial constructor, the use of single operators per iteration, and the effects of incorporating Simulated Annealing and the 2-Opt local search. Unless explicitly specified, results are reported without including 2-Opt local search or electric vehicles.

3.1 Tuning Results

The final parameters obtained after the tuning process are shown in Table 3. The number of iterations, $n_{\text{Iterations}}$, was chosen such that none of the instances showed an improvement of more than 1% in the solution cost beyond this point. Although many instances exhibit very little improvement after the first 40 iterations, we selected 80 iterations because the computational cost per iteration is relatively low.

Global Parameters		
$n_{\text{Iterations}}$	80	Number of ALNS iterations
minSizeNBH	10	Minimum neighborhood size
maxSizeNBH	45	Maximum neighborhood size
updateSpeed	0.8	Weight update speed for adaptive mechanism
startTempControl	0.1	Initial temperature control (w)
coolingRate	0.7	Cooling rate for Simulated Annealing (c)
Operator-specific Parameters		
p	5	Parameter for Worst Removal operator
Regretk	2	Parameter k for Regret Insertion operator
α	0.25	Shaw Removal weight for distance

Table 3: Tuned parameters for the ALNS algorithm.

3.2 Comparison of Final Solutions with Initial Solutions

As shown in Table 4, our solutions improved the cost for all instances. The improvements are particularly significant for the larger instances.

Instance	Initial Solution	Our Results
c202C16	468.7	431.0
lc102	2120.7	679.1
lc108	2443.3	638.5
lc207	2254.3	588.1
lr112	1640.0	1043.0
lr205	2032.7	1240.1
lrc104	1858.2	1137.6
lrc206	2830.9	1380.9
r102C18	264.0	250.4
rc204C16	298.5	298.5

Table 4: Comparison of the initial solution and our results.

3.3 Initial constructor

Instance	Random Insertion	Greedy Insertion	Regret Insertion
c202C16	431.01	431.01	431.01
lc102	667.58	650.00	630.77
lc108	638.54	638.54	594.97
lc207	629.44	629.44	545.24
lr112	1021.38	1042.99	974.29
lr205	1207.62	1218.99	1111.99
lrc104	1183.06	1142.63	1018.42
lrc206	1380.86	1340.84	1317.59
r102C18	250.43	250.43	249.02
rc204C16	317.03	348.60	317.03
	772.70	**769.35**	**719.03**

Table 5: Comparison of initialization methods: Random, Greedy, and Regret Insertion.

Table 5 shows that initialising the solution using the regret insertion method consistently produces the best results on all instances, making it the preferred construction method.

3.4 Single Operators

In it's basic configuration, each iteration picks a pair of destroy and repair operators. It is interesting to also evaluate the performance of each operator taken by itself. In order to do so, we tested all combinations of destroy/repair operator (using the same operator for each iteration)[1].

Instance	Best Combination	Cost	CPU Time	Norm cost	Norm CPU Time
c202C16	(4, 1)	428.1	0	431.0	0
lc102	(3, 2)	570.0	20	679.1	5
lc108	(4, 2)	573.0	5	638.5	9
lc207*	(4, 1)	545.2	5	588.1	27
lr112	(1, 3)	940.6	19	1043.0	6
lr205	(3, 3)	1161.5	54	1240.1	21
lrc104	(1, 3)	1019.9	17	1137.6	5
lrc206	(3, 3)	1235.3	51	1380.9	12
r102C18*	(4, 1)	249.0	0	250.4	0
rc204C16*	(1, 1)	298.5	0	298.5	0

Table 6: Best operator combination for each instance. *These instances have also other combinations of operators with the same cost and equal or greater computation time.

Table 6 shows the best combination of parameters[2] for each instance. The test does not suggest any clear pattern in terms of which combination or which operator has the best performance. It also compares the best combination found to the results of the basic configuration.
Using the best combination consistently produces better results across all instances albeit, it almost always takes significantly more time. It is hard to identify any operator that stands out. All we can say though is that shaw removal never appears in the best combination, suggesting it may not be a very effective way of destroying the solution. We can speculate that this is because, given the little variation in solution it creates due to its nature, it doesn't allow to escape local minima.
Given that no combination stands out over the others, we conclude it is more effective to use a range of operators to solve the problem. There is no effective way of finding the best combination apriori (without testing all of them, which takes significant time) and when this combination finds a better solution it usually comes with a significant increase in computation time.

3.5 Simulated Annealing

Table 7 shows that the Simulated Annealing (SA) algorithm does not always improve the final solution. In several instances, the algorithm without SA achieved slightly better costs. This may occur because ALNS already provides strong diversification through its destroy and repair operators, making the probabilistic acceptance of worse solutions in SA sometimes move the search away from a better solution. SA tends to be more beneficial for larger or more complex instances, as these are more likely to become trapped in local optima.

[1] These tests were run without applying 2-Opt local search to save time.

[2] The first number in parenthesis is the destroy operator (1 = random, 2 = shaw, 3 = worst, 4 = route) and the second one is the repair (1 = random, 2 = greedy, 3 = regret)

Instance	With SA	Without SA
c202C16	431.0	424.9
lc102	679.1	652.4
lc108	638.5	668.7
lc207	588.1	592.1
lr112	1043.0	1028.3
lr205	1240.1	1254.4
lrc104	1137.6	1139.2
lrc206	1380.9	1339.1
r102C18	250.4	250.4
rc204C16	298.5	298.5

Table 7: Comparison of solution costs for ALNS with and without Simulated Annealing (SA).

3.6 2-Opt Local Search Results

Instance	No 2-Opt	After Global Best	Every Iteration	After All Iterations
c202C16	431.0	431.0	431.0	431.0
lc102	679.1	679.1	679.1	679.1
lc108	638.5	638.5	657.3	634.3
lc207	588.1	629.4	589.8	588.1
lr112	1043.0	1028.7	1020.8	1013.1
lr205	1240.1	1207.6	1195.3	1196.8
lrc104	1137.6	1114.0	1143.0	1114.0
lrc206	1380.9	1380.9	1300.9	1340.8
r102C18	250.4	250.4	250.4	250.4
rc204C16	298.5	348.6	298.5	298.5

Table 8: Comparison of solution costs with different applications of 2-Opt.

Table 8 shows that applying 2-Opt after all iterations generally improves the solution slightly or leaves it unchanged. Applying 2-Opt after the global best solution or in every iteration carries the risk of the search getting stuck in a local optimum. In the case of `lrc206`, applying 2-Opt in every iteration produced a lower cost, but for other instances, intermediate applications either had no effect or slightly worsened the solution. Overall, it makes sense to apply 2-Opt only after all iterations to avoid being trapped in a local optimum. Larger routes benefit more from 2-Opt, as longer routes have more opportunities for edge crossings, while short routes with only a few requests are often already nearly 2-optimal.

4 Conclusion

In this project we successfully developed and implemented an Adaptive Large Neighborhood Search (ALNS) heuristic to solve the Pickup and Delivery Problem with Time Windows (PDPTW). The heuristic was constructed using a comprehensive set of seven operators (four destruction and three repair operators), an adaptive weighting mechanism, and a Simulated Annealing (SA). Through rigorous parameter tuning, the algorithm was configured with values in Table 3, this configuration achieved a significant improvement in total route cost over the initial solutions across all tested instances.

No single pair of destruction/repair operators consistently produced the best solution, confirming the need for a diverse and adaptive operator pool to explore complex solution spaces efficiently. This adaptive approach proved to be effective in driving continuous improvement across problem sets. However, changing the initial constructor showed a noticeable improvement, with the Regret Insertion operator producing the best results in all cases. This highlights the importance of starting from a high-quality initial solution, as it can significantly influence the final outcome of the algorithm.
Integration of SA provided additional diversification, although the results showed that it did not always enhance the quality of the final solution. This behaviour suggests that ALNS, by design, already includes sufficient diversification, and SA may introduce excessive randomness when not finely tuned. The experiments with 2-Opt proved valuable as a final post-processing step. Applying 2-Opt after all 80 iterations consistently improved or maintained the solution quality across all instances. Importantly, testing showed that applying 2-Opt during intermediate iterations risked prematurely trapping the search in a local optimum, confirming that it is best used as a final local intensification step.
A major limitation of this work lies in the parameter tuning process, meaning that the calibrated parameters may not generalize well across different datasets or problem structures. The interaction between global parameters (e.g., cooling rate, iteration count) and local parameters (e.g., operator-specific settings) was not fully explored. This simplified approach to tuning may have prevented the algorithm from reaching its best possible configuration. Future work could address this limitation by implementing dynamic tuning, where parameters such as temperature, neighborhood size, or adaptive weights are continuously adjusted during the search based on algorithmic performance. Automated approaches such as reinforcement learning could further improve robustness and adaptability across instances.
The existing Shaw Removal operator is currently limited in design: it employs a simplified relatedness parameter (R) incorporating only the elements of distance and demand. This restricted computation is an simplification of the complete methodology introduced by Ropke & Pisinger (2006). Future developmental work must therefore focus on expanding the Shaw Removal heuristic to incorporate the missing factors present in the established literature, namely temporal relatedness (based on analyzing time window differences) and a full calculation of load differences of the two requests. On the other hand this should come with additional testing, ensuring this actually improves the algorithm leading to better solutions.
Beyond optimizing the current configuration, significant further research can focus on expanding the realism and constraints of the problem model. This involves tackling the currently simplified assumption of a non-restrictive number of available vehicles, necessitating changes to the objective function to prioritize vehicle minimization. Moreover, the implementation of the Electric Vehicle (EV) model can be completed.

References

Demir, E., Bektaş, T., & Laporte, G. (2012, December). An adaptive large neighborhood search heuristic for the Pollution-Routing Problem. *European Journal of Operational Research*, *223*(2), 346–359. Retrieved 2025-10-11, from https://linkinghub.elsevier.com/retrieve/pii/S0377221712004997 doi: 10.1016/j.ejor.2012.06.044

Ropke, S., & Pisinger, D. (2006, November). An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows. *Transportation Science*, *40*(4), 455–472. Retrieved 2025-10-12, from https://pubsonline.informs.org/doi/10.1287/trsc.1050.0135 doi: 10.1287/trsc.1050.0135

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.

B Code

B.1 Main.py

```
# -*- coding: utf-8 -*-
"""
Created on Mon Aug  1 14:34:48 2022

@author: Original template by Rolf van Lieshout
"""

import Problem, Solution, Route
from ALNS import ALNS


testI = "instances/lc207.txt"
problem = Problem.PDPTW.readInstance(testI)
print(problem)
nDestroyOps = 4
nRepairOps = 3
alns = ALNS(problem,nDestroyOps,nRepairOps)
alns.execute()
```

B.2 ALNS.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jul 26 16:28:19 2022

@author: Original template by Rolf van Lieshout
"""
from Solution import Solution
import random, time
import math
import matplotlib.pyplot as plt
import pandas as pd
import os
from Parameters import Parameters

import matplotlib.pyplot as plt
import matplotlib.cm as cm
import numpy as np



class ALNS:
    """
    Class that models the ALNS algorithm.

    Parameters
    ----------
    problem : PDPTW
        The problem instance that we want to solve.
```

```
    nDestroyOps : int
        number of destroy operators.
    nRepairOps : int
        number of repair operators.
    randomGen : Random
        random number generator
    currentSolution : Solution
        The current solution in the ALNS algorithm
    bestSolution : Solution
        The best solution currently found
    bestDistance : int
        Distance of the best solution

    """
    def __init__(self,problem,nDestroyOps,nRepairOps):
        self.problem = problem
        self.nDestroyOps = nDestroyOps
        self.nRepairOps = nRepairOps
        self.destroyOpsWeigths = [(i, 5) for i in range(1, self.nDestroyOps + 1)]
        self.repairOpsWeigths = [(i, 5) for i in range(1, self.nRepairOps + 1)]
        self.randomGen = random.Random(Parameters.randomSeed) #used for reproducibility


    def constructInitialSolution(self):
        """
        Method that constructs an initial solution using random insertion
        """
        self.currentSolution = Solution(self.problem,list(),list(),list(self.problem.requests.copy()))
        #self.currentSolution.executeRandomInsertion(self.randomGen)
        self.currentSolution.executeGreedyInsertion(self.randomGen)
        #self.currentSolution.executeRegretInsertion(self.randomGen)
        self.currentSolution.computeDistance()
        self.bestSolution = self.currentSolution.copy()
        self.bestDistance = self.currentSolution.distance
        ###
        w = Parameters.startTempControl
        z = self.bestDistance
        self.temperature = - (w * z) / math.log(0.5) # P = e ** (-w.z)/tstart  so tstart = -(w *z) / ln(0.5)

        print(self.temperature)

        ###
        print("Created initial solution with distance: "+str(self.bestDistance))

    def execute(self):
        """
        Method that executes the ALNS
        """
        cost = []
        costcu = []
        starttime = time.time() # get the start time
        self.constructInitialSolution()
        for i in range(Parameters.nIterations):
```

```
            #copy the current solution
            self.tempSolution = self.currentSolution.copy()
            #decide on the size of the neighbourhood
            sizeNBH = self.randomGen.randint(Parameters.minSizeNBH,Parameters.maxSizeNBH)
            #decide on the destroy and repair operator numbers
            if not Parameters.overrideOpr:
                destroyOpNr = self.determineDestroyOpNr()
                repairOpNr = self.determineRepairOpNr()
            elif Parameters.overrideOpr:
                destroyOpNr = Parameters.destroy
                repairOpNr = Parameters.repair
            #execute the destroy and the repair and evaluate the result
            self.destroyAndRepair(destroyOpNr, repairOpNr, sizeNBH);
            self.tempSolution.computeDistance()
            self.iterationPrint(i, destroyOpNr, repairOpNr, sizeNBH)
            #print("Iteration "+str(i)+": (destroy: " + str(destroyOpNr) + ", repair: " + str(repairOpNr) + ", NHB size: " + str(sizeNBH) + ") Found solution with distance: "+str(self.tempSolution.distance))
            #self.tempSolution.print()
            #determine if the new solution is accepted
            state = self.checkIfAcceptNewSol()
            #update the ALNS weights
            cost.append((i,self.tempSolution.distance))
            costcu.append((i,self.bestSolution.distance))
            self.updateWeights(state, destroyOpNr, repairOpNr)
        endtime = time.time() # get the end time
        self.cpuTime = round(endtime-starttime)
        print("Terminated. Final distance: "+str(self.bestSolution.distance)+", cpuTime: "+str(self.cpuTime)+" seconds")
        #self.plot_routes()

        # self.drawGraph(cost)
        # self.drawGraph(costcu)

    def drawGraph(self,data):
        x = [item[0] for item in data]
        y = [item[1] for item in data]


        results = pd.DataFrame({
            'Iter': x,
            'Cost': y
            })
        fileName = "log/" +  str(self.problem.name)
        print(fileName)
        results.to_csv(fileName, index = False)
        figureName = fileName + ".png"
        plt.plot(x,y,marker='o')
        plt.show()
        plt.savefig(figureName)
        plt.close('all')
```

```
    def iterationPrint(self, iterationNr, destroyOpNr, repairOpNr, sizeNBH):
        i = str(iterationNr)
        destroyOp = str(destroyOpNr)
        destroyW = str(round(self.destroyOpsWeigths[destroyOpNr-1][1], 2))
        repariOp = str(repairOpNr)
        repairW = str(round(self.repairOpsWeigths[repairOpNr-1][1], 2))
        sizeNBH = str(sizeNBH)
        distance = str(self.tempSolution.distance)


        message = "Iteration " + i + ": (destroy: " + destroyOp + " (" + destroyW + "), repair: " + repariOp + " (" + repairW + "), NBH size: " + sizeNBH + "). Found solution with distance: " + distance
        print(message)


    def checkIfAcceptNewSol(self):
        """
        Method that checks if we accept the newly found solution
        """



        state = "Rejected"
        #if we found a global best solution, we always accept

        if self.tempSolution.distance<self.bestDistance:
            if Parameters.maketwoOpt:
                self.currentSolution.ApplyTwoOpt()
            self.bestDistance = self.tempSolution.distance
            self.bestSolution = self.tempSolution.copy()
            self.currentSolution = self.tempSolution.copy()
            state = "Global Best"
            #self.tempSolution.ApplyTwoOpt()
            print("Found new global best solution.\n")


        #currently, we  accept better solution
        if self.tempSolution.distance<self.currentSolution.distance:

            self.currentSolution = self.tempSolution.copy()
            state = "Better Sol"

        # simulated annealing
        elif random.random() < math.e ** -((self.tempSolution.distance - self.currentSolution.distance)/ self.temperature):
            self.currentSolution = self.tempSolution.copy()
            state = "Accepted"
            print("Accepeted the worse soulution")
            #print(self.temperature)

        self.temperature = self.temperature * Parameters.coolingRate

        return state
```

```python
    def updateWeights(self, state, chosenDestroyOp, chosenRepOp):
        """
        Method that updates the weights of the destroy and repair operators
        """
        reward = Parameters.reward[state]
        updateSpeed = Parameters.updateSpeed

        # Update destroy weights
        oldWeight_d = self.destroyOpsWeigths[chosenDestroyOp-1][1]
        newWeight_d = updateSpeed*oldWeight_d + (1-updateSpeed)*reward

        self.destroyOpsWeigths[chosenDestroyOp-1] = (self.destroyOpsWeigths[chosenDestroyOp-1][0], newWeight_d)

        # Update repair weights
        oldWeight_r = self.repairOpsWeigths[chosenRepOp-1][1]
        newWeight_r = updateSpeed*oldWeight_r + (1-updateSpeed)*reward

        self.repairOpsWeigths[chosenRepOp-1] = (self.repairOpsWeigths[chosenRepOp-1][0], newWeight_r)



    def determineDestroyOpNr(self):
        """
        Method that determines the destroy operator that will be applied.
        Currently we just pick a random one with equal probabilities.
        Could be extended with weights
        """
        selectedOpNr = self.randomGen.choices([t[0] for t in self.destroyOpsWeigths], weights=[t[1] for t in self.destroyOpsWeigths], k = 1 )[0]
        return selectedOpNr #self.randomGen.randint(1, self.nDestroyOps)

    def determineRepairOpNr(self):
        """
        Method that determines the repair operator that will be applied.
        Currently we just pick a random one with equal probabilities.
        Could be extended with weights
        """
        selectedOpNr = self.randomGen.choices([t[0] for t in self.repairOpsWeigths], weights=[t[1] for t in self.repairOpsWeigths], k = 1 )[0]
        return selectedOpNr

    def destroyAndRepair(self,destroyHeuristicNr,repairHeuristicNr,sizeNBH):
        """
        Method that performs the destroy and repair. More destroy and/or
        repair methods can be added

        Parameters
        ----------
        destroyHeuristicNr : int
            number of the destroy operator.
        repairHeuristicNr : int
            number of the repair operator.
        sizeNBH : int
```

```python
            size of the neighborhood.

        """
        #perform the destroy
        if destroyHeuristicNr == 1:
            self.tempSolution.executeRandomRemoval(sizeNBH,self.randomGen)
        elif destroyHeuristicNr == 2:
            self.tempSolution.executeShawRemoval(sizeNBH, self.randomGen)
        elif destroyHeuristicNr == 3:
            self.tempSolution.executeWorstReomval(sizeNBH, self.randomGen)
        elif destroyHeuristicNr == 4:
            self.tempSolution.executeRouteRemoval(self.randomGen)
        #perform the repair
        if repairHeuristicNr == 1:
            self.tempSolution.executeRandomInsertion(self.randomGen)
        elif repairHeuristicNr == 2:
            self.tempSolution.executeGreedyInsertion(self.randomGen)
        elif repairHeuristicNr == 3:
            self.tempSolution.executeRegretInsertion(self.randomGen)

    def plot_routes(self):
            """
            Plots the routes
            """
            plt.figure(figsize=(12, 12))

            num_routes = len(self.bestSolution.routes) # using rainbow is the suggestion by
            ↪  AI, before that the colors was repated sometimes
            colors = cm.rainbow(np.linspace(0, 1, num_routes))

            for route,color in zip(self.bestSolution.routes, colors):
                x_coords = [loc.xLoc for loc in route.locations]
                y_coords = [loc.yLoc for loc in route.locations]
                plt.plot(x_coords, y_coords, marker='o',color = color)


            plt.plot(self.problem.depot.xLoc, self.problem.depot.yLoc, 'p', markersize=15)




            plot_filename = f"log/{os.path.basename(self.problem.name)}_routes.png"
            plt.savefig(plot_filename)
            print(f"Route plot saved to {plot_filename}")
            plt.close('all')
            #plt.show()



```

B.3 Parameters.py

```python
import random
class Parameters:
```

```
    """
    Class that holds all the parameters for ALNS
    """

    maketwoOpt = False


    nIterations = 80  #number of iterations of the ALNS

    minSizeNBH = 10      #minimum neighborhood size CALIBRATED
    maxSizeNBH = 45     #maximum neighborhood size CALIBRATED

    randomSeed = 1      #value of the random seed
    reward = {
        "Global Best": 10,
        "Better Sol": 8,
        "Accepted": 5,
        "Rejected": 1
    }
    updateSpeed = 0.8 # For adaptive CALIBRATED

    #can add parameters such as cooling rate etc.
    startTempControl = 0.1 # Calibrated  # this means if will accept the solotions with 10%
    ↪ highes cost with 50% Prob
    coolingRate = 0.7 #     updateSpeed = 0.8 # For adaptive

    useBattery = True
    # ------------------------- #

    p = 5 # Calibrated
    Regretk = 2 # Calibrated

    # For shaw removal
    alpha = 0.25 # Calibrated

    # To override operators:
    overrideOpr = False
    destroy = 2
    repair = 2
```

B.4 Problem.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jul 26 12:41:37 2022

@author: Original template by Rolf van Lieshout
"""

import numpy as np
import math

class Request:
    """
```

```python
    Class that represents a request

    Attributes
    ----------
    pickUpLoc : Location
        The pick-up location.
    deliveryLoc : Location
        The delivery location.
    ID : int
        id of request.

    """
    def __init__(self,pickUpLoc,deliveryLoc,ID):

        self.pickUpLoc = pickUpLoc
        self.deliveryLoc = deliveryLoc
        self.ID = ID

class Location:
    """
    Class that represents either (i) a location where a request should be picked up
    or delivered or (ii) the depot
    Attributes
    ----------
    requestID : int
        id of request.
    xLoc : int
        x-coordinate.
    yLoc : int
        y-coordinate.
    demand : int
        demand quantity, positive if pick-up, negative if delivery
    startTW : int
        start time of time window.
    endTW : int
        end time of time window.
    servTime : int
        service time.
    typeLoc : int
        1 if pick-up, -1 if delivery, 0 if depot
    nodeID : int
        id of the node, used for the distance matrix
    """
    def __init__(self,requestID,xLoc,yLoc,demand,startTW,endTW,servTime,typeLoc,nodeID):

        self.requestID = requestID
        self.xLoc = xLoc
        self.yLoc = yLoc
        self.demand = demand
        self.startTW = startTW # start Time Window
        self.endTW = endTW
        self.servTime = servTime
        self.typeLoc = typeLoc
        self.nodeID = nodeID

```

```
    def __str__(self):
        """
        Method that prints the location
        """
        return f"({self.requestID},{self.typeLoc})"


    def getDistance(l1,l2):
        """
        Method that computes the euclidian distance between two locations
        """
        dx = l1.xLoc-l2.xLoc
        dy = l1.yLoc-l2.yLoc
        return math.sqrt(dx**2+dy**2)

class PDPTW:
    """
    Class that represents a pick-up and delivery problem with time windows
    Attributes
    ----------
    name : string
        name of the instance.
    requests : List of Requests
        The set containing all requests.
    depot : Location
        the depot where all vehicles must start and end.
    locations : Set of Locations
        The set containing all locations
     distMatrix : 2D array
         matrix with all distances between cities
    capacity : int
        capacity of the vehicles

    """
    def __init__(self,name,requests,depot,vehicleCapacity, vehicleBattery):
        self.name = name
        self.requests = requests
        self.depot = depot
        self.capacity = vehicleCapacity
        self.battery = vehicleBattery
        ##construct the set with all locations
        self.locations = set()
        self.locations.add(depot)
        for r in self.requests:
            self.locations.add(r.pickUpLoc)
            self.locations.add(r.deliveryLoc)

        #compute the distance matrix
        self.distMatrix = np.zeros((len(self.locations),len(self.locations))) #init as nxn matrix
        for i in self.locations:
            for j in self.locations:
                distItoJ = Location.getDistance(i,j)
                self.distMatrix[i.nodeID,j.nodeID] = distItoJ
        self.distMatrix_Max = np.max(self.distMatrix)
```

```
        self.distMatrix_Min = np.min(self.distMatrix)

    def __str__(self):
        return f" PDPTW problem {self.name} with {len(self.requests)} requests and a vehicle capacity of {self.capacity}"


    def readInstance(fileName):
        """
        Method that reads an instance from a file and returns the instancesf
        """
        f = open(fileName)
        requests = list()
        unmatchedPickups = dict()
        unmatchedDeliveries = dict()
        nodeCount = 0
        requestCount = 1
        for line in f.readlines()[1:-6]:
            asList = []
            n  = 13 #columns start every 13 characters
            for index in range(0, len(line), n):
                asList.append(line[index : index + n].strip())


            lID = asList[0]
            x = int(asList[2][:-2]) #need to remove ".0" from the string
            y = int(asList[3][:-2])
            if lID.startswith("D"):
                #it is the depot
                depot = Location(0,x,y,0,0,0,0,0,nodeCount)
                nodeCount += 1
            elif lID.startswith("C"):
                # it is a location
                lType = asList[1]
                demand = int(asList[4][:-2])
                startTW = int(asList[5][:-2])
                endTW = int(asList[6][:-2])
                servTime = int(asList[7][:-2])
                partnerID = asList[8]
                if lType == "cp":
                    #it is a pick-up
                    if partnerID in unmatchedDeliveries:
                        deliv = unmatchedDeliveries.pop(partnerID)
                        pickup = Location(deliv.requestID, x, y, demand, startTW, endTW, servTime, 1, nodeCount)
                        nodeCount += 1
                        req = Request(pickup,deliv,deliv.requestID)
                        requests.append(req)
                    else:
                        pickup = Location(requestCount, x, y, demand, startTW, endTW, servTime, 1, nodeCount)
                        nodeCount += 1
                        requestCount += 1
                        unmatchedPickups[lID] = pickup
                elif lType == "cd":
```

```
                    #it is a delivery
                    if partnerID in unmatchedPickups:
                        pickup = unmatchedPickups.pop(partnerID)
                        deliv = Location(pickup.requestID, x, y, demand, startTW, endTW,
                        ↪  servTime, -1, nodeCount)
                        nodeCount += 1
                        req = Request(pickup,deliv,pickup.requestID)
                        requests.append(req)
                    else:
                        deliv = Location(requestCount, x, y, demand, startTW, endTW,
                        ↪  servTime, -1, nodeCount)
                        nodeCount += 1
                        requestCount += 1
                        unmatchedDeliveries[lID] = deliv

        #sanity check: all pickups and deliveries should be matched
        if len(unmatchedDeliveries)+len(unmatchedPickups)>0:
            raise Exception("Not all matched")

        # read the vehicle capacity
        f = open(fileName)
        print("helo")
        print(f)
        capLine = f.readlines()[-4]
        capacity = int(capLine[-7:-3].strip())

        f = open(fileName)
        batLine = f.readlines()[-5]
        battery = float(batLine[-7:-1].strip())
        print(battery)
        return PDPTW(fileName,requests,depot,capacity, battery)
```

B.5 Route.py

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jul 28 17:10:03 2022

@author: Original template by Rolf van Lieshout
"""
import sys
from Problem import Location
from Parameters import Parameters

class Route:
    """
    Class used to represent a route

    Parameters
    ----------
    locations : list of locations
        the route sequence of locations.
    requests : list of requests
        the requests served by the route
```

```
    problem : PDPTW
        the problem instance, used to compute distances.
    feasible : boolean
        true if route respects time windows, capacity and precedence
    distance : int
        total distance driven, extremely large number if infeasible
    """
    def __init__(self,locations,requests,problem):
        self.locations = locations
        self.requests = requests
        self.problem = problem
        #check the feasibility and compute the distance
        self.feasible = self.isFeasible()
        if self.feasible:
            self.distance = self.computeDistance()
        else:
            self.distance = sys.maxsize #extremely large number

    def computeDistance(self):
        """
        Method that computes and returns the distance of the route
        """
        totDist = 0
        for i in range(1,len(self.locations)-1):
            prevNode = self.locations[i-1]
            curNode = self.locations[i]
            dist = self.problem.distMatrix[prevNode.nodeID][curNode.nodeID]
            totDist += dist
        return totDist

    def __str__(self):
        """
        Method that prints the route
        """
        toPrint = "Route "
        for loc in self.locations:
            toPrint += loc.__str__()
        toPrint += f" dist={self.distance}"
        return toPrint

    def isFeasible(self):
        """
        Method that checks feasbility. Returns True if feasible, else False
        """
        #route should start and end at the depot
        if self.locations[0]!=self.problem.depot or self.locations[-1]!=self.problem.depot:
            return False

        curTime = 0 #current time
        curLoad = 0 #current load in vehicle
        curNode = self.locations[0] #current node
        pickedUp = set() #set with all requests that we picked up, used to check precedence

        #totDistance = 0
        curCharge = self.problem.battery
```

```
        #iterate over route and check feasibility of time windows, capacity and precedence
        for i in range(1,len(self.locations)-1):
            prevNode = self.locations[i-1]
            curNode = self.locations[i]
            dist = self.problem.distMatrix[prevNode.nodeID][curNode.nodeID]
            curTime = max(curNode.startTW, curTime + prevNode.servTime + dist)
            #totDistance += dist
            #check if time window is respected
            if curTime>curNode.endTW:
                return False
            #check if capacity not exceeded

            curLoad += curNode.demand
            if curLoad>self.problem.capacity:
                return False
            #check if vehicle has enough charge
            if Parameters.useBattery:
                curCharge = curCharge - dist
                if curCharge < 0:
                   return False
            #check if we don't do a delivery before a pickup
            if curNode.typeLoc == 1:
                #it is a pickup
                pickedUp.add(curNode.requestID)
            else:
                #it is a delivery
                #check if we picked up the request
                if curNode.requestID not in pickedUp:
                    return False
                pickedUp.remove(curNode.requestID)

        #finally, check if all pickups have been delivered
        if len(pickedUp)>0:
            return False
        return True

    def removeRequest(self,request):
        """
        Method that removes a request from the route.
        """
        #remove the request, the pickup and the delivery
        self.requests.remove(request)
        self.locations.remove(request.pickUpLoc)
        self.locations.remove(request.deliveryLoc)
        #the distance changes, so update
        self.distance = self.computeDistance()

    def copy(self):
        """
        Method that returns a copy of the route
        """
        locationsCopy = self.locations.copy()
        requestsCopy = self.requests.copy()
        return Route(locationsCopy,requestsCopy,self.problem)

```

```python
    def greedyInsert(self,request):
        """
        Method that inserts the pickup and delivery of a request at the positions
        that give the shortest total distance. Returns best route.

        Parameters
        ----------
        request : Request
            the request that should be inserted.

        Returns
        -------
        bestInsert : Route
            Route with the best insertion.

        """
        requestsCopy = self.requests.copy()
        requestsCopy.append(request)

        minDist = sys.maxsize #initialize as extremely large number
        bestInsert = None
        #iterate over all possible insertion positions for pickup and delivery
        for i in range(1,len(self.locations)):
            for j in range(i+1,len(self.locations)+1): #delivery after pickup
                locationsCopy = self.locations.copy()
                locationsCopy.insert(i,request.pickUpLoc)
                locationsCopy.insert(j,request.deliveryLoc)
                afterInsertion = Route(locationsCopy,requestsCopy,self.problem)
                #check if insertion is feasible
                if afterInsertion.feasible:
                    #check if cheapest
                    if afterInsertion.distance<minDist:
                        bestInsert = afterInsertion
                        minDist = afterInsertion.distance

        return bestInsert, minDist


    def twoOpt(self):
        """
        applies the 2-opt heuristic to the route.
        """
        improved = True
        current_route = self
        while improved:
            improved = False
            best_distance = current_route.distance
            for i in range(1, len(current_route.locations) - 2):
                for j in range(i + 1, len(current_route.locations) - 1):

                    new_locations = current_route.locations[:i] + current_route.locations[i:j+1][::-1] + current_route.locations[j+1:]
                    new_route = Route(new_locations, current_route.requests, current_route.problem)

```

```
                    # If the new route is feasible and better
                    if new_route.isFeasible() and new_route.distance < best_distance: # This may resaults to some routes which are not two opt because the two opt version of them are not feasible
                        current_route = new_route
                        best_distance = new_route.distance
                        improved = True


                        break
                if improved:
                    break
        return current_route
```

B.6 Solution.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jul 26 13:54:49 2022

@author: Original template by Rolf van Lieshout
"""
import numpy as np
import sys
from Route import Route
from Problem import Location, PDPTW
from Parameters import Parameters




class Solution:
    """
    Method that represents a solution tot the PDPTW

    Attributes
    ----------
    problem : PDPTW
        the problem that corresponds to this solution
    routes : List of Routes
         Routes in the current solution
    served : List of Requests
        Requests served in the current solution
    notServed : List of Requests
         Requests not served in the current solution
    distance : int
        total distance of the current solution
    """
    def __init__(self,problem,routes,served,notServed):
        self.problem = problem
        self.routes = routes
        self.served = served
        self.notServed = notServed

```

```python
    def computeDistance(self):
        """
        Method that computes the distance of the solution
        """
        self.distance = 0
        for route in self.routes:
            self.distance += route.distance

    def __str__(self):
        """
        Method that prints the solution
        """
        nRoutes = len(self.routes)
        nNotServed = len(self.notServed)
        toPrint = f"Solution with {nRoutes} routes and {nNotServed} unserved requests: "
        for route in self.routes:
            toPrint+= route.__str__()

    def executeRandomRemoval(self,nRemove,random):
        """
        Method that executes a random removal of requests

        This is destroy method number 1 in the ALNS

        Parameters
        ----------
        nRemove : int
            number of requests that is removed.

        Parameters
        ----------
        randomGen : Random
            Used to generate random numbers

        """


        for i in range(nRemove):
            #terminate if no more requests are served
            if len(self.served)==0:
                break
            #pick a random request and remove it from the solutoin
            req = random.choice(self.served)
            self.removeRequest(req)



    def executeShawRemoval(self, nRemove, random):
        """
        Method that executes Shaw Removal Heuristic: it removes requests that are somewhat similar. This is a variation of the method proposed by Ropke et al. (2006).
        It only considers distance and demand as parameters to evaluate relatedness.

        It's destroy method number 2 in the ALNS

```

```
        Parameters
        ----------
        nRemove : int
            number of requests that are removed.
        randomGen : random
            Used to generate random numbers

        """

        if len(self.served) == 0:
            return
        # Pick a random request (then find similar ones)
        req = random.choice(self.served)
        candidates = self.evaluateRelatedness(req)
        # print(candidates[0][1])
        # print(candidates[1][1])
        for i in range(nRemove):
            if len(self.served) == 0:
                break
            self.removeRequest(candidates[i][1])

    def evaluateRelatedness(self, req):
        """
        Method taht calculates a relatedness parameter between a reference request (req) and all other requests in the problem, returning a list of requests ordered from greatest to lowest relatedness.

        It uses only 2 parameters: distance and demand
        """

        # Pick a random request (then find similar ones)

        self.removeRequest(req)

        relatedness = []

        for request in self.served:
            # Distance component
            pickUpDist = Location.getDistance(req.pickUpLoc, request.pickUpLoc)
            requestDist = Location.getDistance(req.deliveryLoc, request.deliveryLoc)
            R_dist = (pickUpDist+requestDist)  / self.problem.distMatrix_Max

            # Demand component (normalized)
            req_demand = req.pickUpLoc.demand
            request_demand = request.pickUpLoc.demand

            R_demand = abs(req_demand - request_demand) / self.problem.capacity # Assumes no loc has demand greater than vehicle capacity. Should not matter, since we are using difference between demands anyway, so this should alsways be between 0 and 1.
            alpha = Parameters.alpha
            beta = 1-alpha
            # Calculate R (relatedness)
            R =  alpha*R_dist + beta*R_demand # relatedness parameter
            relatedness.append((req, request, R))
```

```
            # print("Printing relatedness")
            # print(relatedness[request])
        relatedness.sort(key=lambda x: x[2], reverse = True) # Sort all requests based on relatedness
        return relatedness


    def executeWorstReomval(self,nReomve, random):
        """
        Method that executes Worst Removal Heuristic: it removes the requests that appear to be placed in the wrong position in the solution. This is a variation of the method proposed by Ropke et al. (2006).


        It's destroy method number 3 in the ALNS

        Parameters
        ----------
        nRemove : int
            number of requests that are removed.
        randomGen : random
            Used to generate random numbers

        """

        if len(self.served) == 0:
            return


        while nReomve > 0:
            cost = []
            if len(self.served) == 0:
                break
            for req in self.served: # to find which route is now serving this requset
                routefound = None
                for route in self.routes:
                    if req in route.requests:
                        routefound = route
                        break
                if routefound is None: # Code should not reach here
                    continue
                # This can be impoved for efficency
                # insted of calcualting the whole tour, it is possible to calculate the two new line and minus it from the orginal

                orginalCost = routefound.distance
                temp = routefound.copy()
                temp.removeRequest(req)
                newCost =temp.distance
                deltaCost = orginalCost - newCost
                cost.append((req,deltaCost))


            # randomization controlled by the parameter p
            p = Parameters.p
```

```python

            cost.sort(key=lambda x: x[1] , reverse=True)   # sort by form worst based on the delta cost
            # The random removal
            randomN = random.random()
            reqNumber = int(len(cost) * (randomN ** p))

            self.removeRequest(cost[reqNumber][0])
            nReomve -= 1

    def executeRouteRemoval(self, random):
        """
        Method that removes a number of routes from the solution: it removes all requests part of that route
        It will remove a random number between 1 and 75% of the number of routes currenlty in the solutions. This is to avoid it destroying the whole solution

        It's destroy method number 4 in the ALNS

        Parameters
        ----------
        nRemove : int
            number of requests that are removed.
        randomGen : random
            Used to generate random numbers
        """

        removeRange = int(0.75*len(self.routes))
        removeRange = max(2, removeRange)
        n = random.choice(range(1,removeRange))

        chosenRoutes = random.sample(self.routes, n)
        for route in chosenRoutes:
            for request in route.requests:
                self.removeRequest(request)



    def removeRequest(self,request):
        """
        Method that removes a request from the solution
        """
        #iterate over routes to find in which route the request is served
        for route in self.routes:
            if request in route.requests:
                #remove the request from the route and break from loop
                route.removeRequest(request)
                break
        #update lists with served and unserved requests
        self.served.remove(request)
        self.notServed.append(request)

    def copy(self):
        """
        Method that creates a copy of the solution and returns it
```

```
        """
        #need a deep copy of routes because routes are modifiable
        routesCopy = list()
        for route in self.routes:
            routesCopy.append(route.copy())
        copy = Solution(self.problem,routesCopy,self.served.copy(),self.notServed.copy())
        copy.computeDistance()
        return copy

    def executeRandomInsertion(self,randomGen):
        """
        Method that randomly inserts the unserved requests in the solution

        This is repair method number 1 in the ALNS

        Parameters
        ----------
        randomGen : Random
            Used to generate random numbers

        """

        #iterate over the list with unserved requests
        while len(self.notServed)>0:
            #pick a random request
            req = randomGen.choice(self.notServed)

            #keep track of routes in which req could be inserted
            potentialRoutes = self.routes.copy()
            inserted = False
            while len(potentialRoutes)>0:
                #pick a random route

                randomRoute = randomGen.choice(potentialRoutes)

                afterInsertion, _ = randomRoute.greedyInsert(req)
                if afterInsertion==None:
                    #insertion not feasible, remove route from potential routes
                    potentialRoutes.remove(randomRoute)
                else:
                    #insertion feasible, update routes and break from while loop
                    inserted = True
                    #print("Possible")
                    self.routes.remove(randomRoute)
                    self.routes.append(afterInsertion)
                    break

            # if we were not able to insert, create a new route
            if not inserted:
                #create a new route with the request
                locList =
                ↪  [self.problem.depot,req.pickUpLoc,req.deliveryLoc,self.problem.depot]
                newRoute = Route(locList,[req],self.problem)
                self.routes.append(newRoute)
            #update the lists with served and notServed requests
```

```
            self.served.append(req)
            self.notServed.remove(req)

    def executeGreedyInsertion(self, randomGen):
        """
        Method that inserts unserved requests in the solution using a basic greedy heuristic.
        It looks for the best overall position to insert each requests.

        This is repair method number 2 in the ANLS.

        Parameters
        ----------
        randomGen : Random
            Used to generate random numbers
        """

        while len(self.notServed) > 0:
            reqBank = []
            bestRequest = None
            bestRoute = None
            bestDist = sys.maxsize
            inserted = False
            for route in self.routes:
                candidateRequest = None
                candidateRoute = None
                candidateDist = sys.maxsize
                for req in self.notServed:

                    newRoute, dist = route.greedyInsert(req)

                    if newRoute == None:
                        reqBank.append(req)

                        continue
                    elif dist<candidateDist:
                        candidateRequest = req
                        candidateRoute = newRoute
                        candidateDist = dist
                if candidateRoute==None:
                    continue
                if candidateDist < bestDist:
                    inserted = True
                    routeToRemove = route
                    bestRequest = candidateRequest
                    bestRoute = candidateRoute
                    bestDist = candidateDist
            if inserted==True:
                self.routes.remove(routeToRemove)
                self.routes.append(bestRoute)
                self.served.append(bestRequest)
                self.notServed.remove(bestRequest)
            if bestRequest == None:
                #Impossible to insert in existing routes, create new route:
                #print(reqBank)
```

```
                req = randomGen.choice(self.notServed)

                locList = [self.problem.depot,req.pickUpLoc,req.deliveryLoc,self.problem.depot]
                newRoute = Route(locList,[req],self.problem)
                self.routes.append(newRoute)
                self.served.append(req) #
                self.notServed.remove(req)
            #update the lists with served and notServed requests
            # I think we should also apend to served and remove from not serves here


    def executeRegretInsertion(self, randomG):
        """
        Method that inserts unserved requests in the solution using the Regret-k heuristic.
        The regret heuristic tries to improve upon the basic greedy heuristic by incorporating a kind of look ahead
        information when selecting the request to insert.
        k is set to 2 right now.

        This is repair method number 3 in the ANLS.



        Parameters
        ----------
        randomGen : Random
            Used to generate random numbers

        """
        k = Parameters.Regretk # we can change this,
        while len(self.notServed) > 0:
            bestRequest = None
            bestRoute = None
            bestRegret = -sys.maxsize # to keep track of the largest Regret
            bestCost = sys.maxsize
            routeToRemove = None

            for req in self.notServed:
                costs = []

                for route in self.routes:
                    newRoute, dist = route.greedyInsert(req)
                    if newRoute is not None:
                        costs.append((dist,newRoute,route))
                        # newRoute is after insertion
                        # roure is before

                if len(costs) == 0:
                    continue

                costs.sort(key=lambda x:x[0]) # sort based on dist

                regretCost = 0

```

```
                for j in range(1,min(k,len(costs))):
                    regretCost = regretCost + costs[j][0] - costs[0][0] # regret cost between best and second option
                if regretCost > bestRegret or (regretCost == bestRegret and costs[0][0] < bestCost): # Ties are broken by selecting the request with best insertion cost
                        bestRegret = regretCost
                        bestRequest = req
                        bestRoute = costs[0][1]  # best route to insert
                        routeToRemove = costs[0][2]
                        #print(cost[0])
                        bestCost = costs[0][0] # we need this for tie breaks



            if bestRequest is None:
                req = randomG.choice(self.notServed)
                locList = [self.problem.depot, req.pickUpLoc, req.deliveryLoc, self.problem.depot]
                newRoute = Route(locList, [req], self.problem)
                self.routes.append(newRoute)
                self.served.append(req)
                self.notServed.remove(req)
            else:
                self.routes.remove(routeToRemove)
                self.routes.append(bestRoute)
                self.served.append(bestRequest)
                self.notServed.remove(bestRequest)
            #break
        #print(cost)
        #print(costs[0])


    def ApplyTwoOpt(self):

        routes = []

        for route in self.routes:
            twoOpt = route.twoOpt()

            if twoOpt.distance < route.distance:
                print("Two Opt Made")

            routes.append(twoOpt)

        self.routes = routes
        self.computeDistance()
```

B.7 experiment.py

```python
import Problem, Solution, Route
from ALNS import ALNS
from Parameters import Parameters
import numpy as np
import os
import time
import pandas as pd


a = 0
b = 45
s = 5

values = np.arange(a, b, s).tolist()

#tempControl = [0.05 ,0.1, 0.2, 0.3, 0.5, 0.8]
#colling_rate = [0.1,0.3,0.5,0.6,0.7,0.8,0.9]
instance_dir = "Instances"

# Get a list of all files in the Instances directory
instance_files = os.listdir(instance_dir)





FinalResulats = []
for instance_file in instance_files: # this is AI genrated
# Construct the full path to the instance file
    for i in values:
        results = []

        for randomSeed in range(3):
            testI = os.path.join(instance_dir, instance_file)

            Parameters.randomSeed = randomSeed
            Parameters.minSizeNBH = i

            problem = Problem.PDPTW.readInstance(testI)
            print(problem)
            nDestroyOps = 4
            nDestroyOps = 4
            nRepairOps = 3
            alns = ALNS(problem,nDestroyOps,nRepairOps)
            starttime = time.time()
            alns.execute()
            #print(alns.bestSolution.distance)




            results.append({'distance': alns.bestSolution.distance})

```

```
            break

        #print(results)
        avg_distance = np.mean([res['distance'] for res in results])
        #avg_time = np.mean([res['time'] for res in results])

        FinalResulats.append({
    'instance' : os.path.basename(alns.problem.name),
    'minNBH': i,
    'average best distance': alns.bestSolution.distance,
    #'average time': avg_time
    })

#print(FinalResulats)

df = pd.DataFrame(FinalResulats)
df.to_csv("minNBHsize.csv")
#print(df)
```