Department of Management Science & Technology

MSc in Business Analytics

CluVRP Solver Documentation

By

Michail Kalligas

Student ID Number: f2822103

Name of Supervisor: Emmanouil Zachariadis

January 2023

Athens, Greece

Table of Contents

1. How to use

1.1. Python Interpreter

The code was run with Python 3.9.13

1.2. Necessary modules

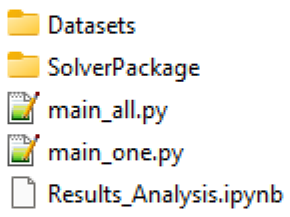
To run the code the below modules need to be installed:

1) pandas
2) tqdm
3) numpy
4) matplotlib
5) PyInquirer (This module is not compatible with Jupyter Lab. It is used to create a user-interface for instance selection. It works only in a cmd environment. If the user wants to avoid installing it, he should delete the import statement in SolverPackage\main_functions.py, and follow the commented instructions in main_one.py & main_all.py)
6) scipy
7) scikit-learn

1.3. How to run

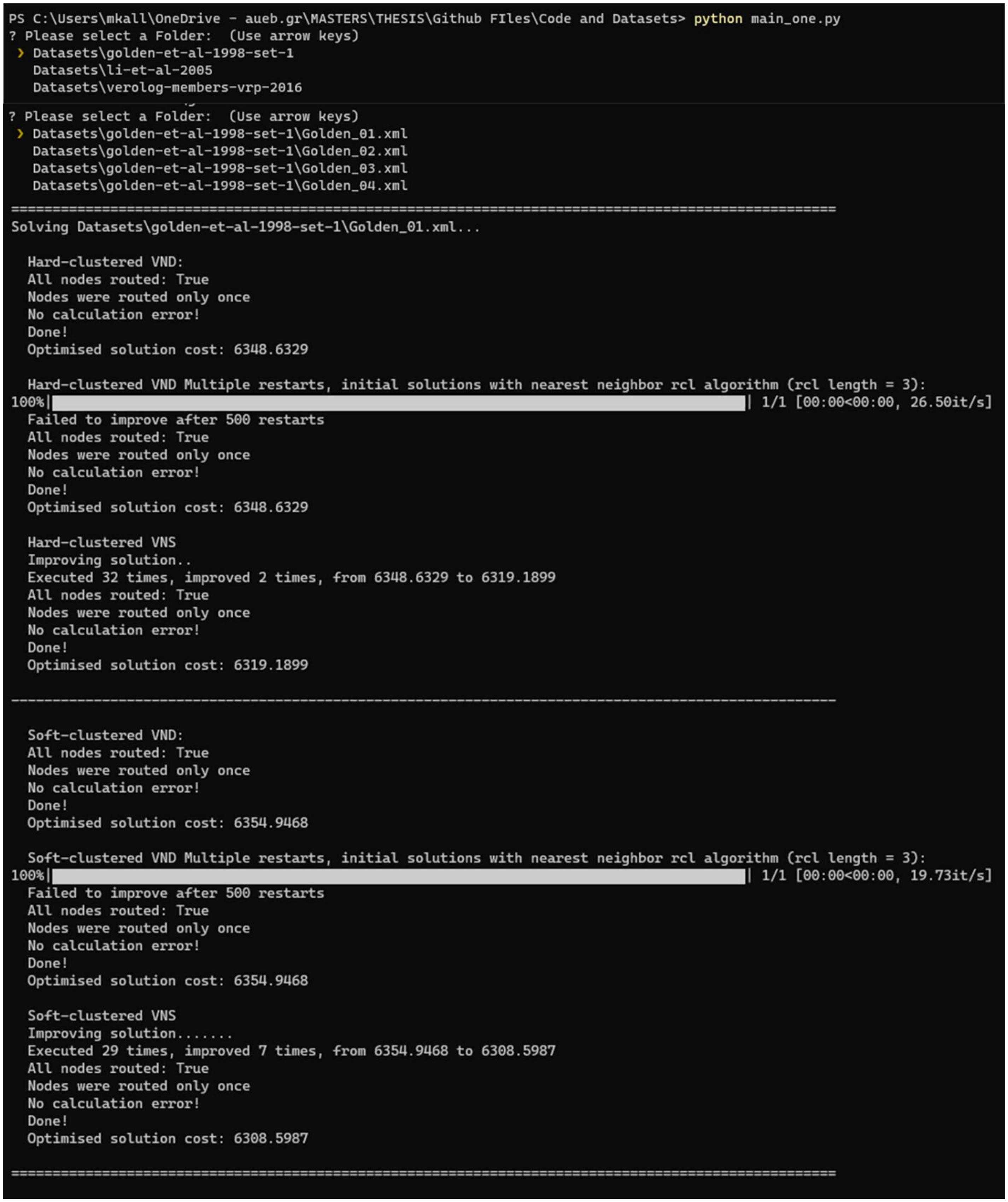
The user should run main_one.py if he wants to solve a single instance, or main_all.py if he wants to solve an entire instance set. The files should be in the same folder as depicted in Figure 1 below.

Figure 1 - Code files


Datasets
SolverPackage
main_all.py
main_one.py
Results_Analysis.ipynb

If the user wants to run the program with a user interface PyInquirer should be installed, and main_one.py or main_all.py should be ran from cmd as shown in Figure 2 below.

Figure 2 - Execution example



```
PS C:\Users\mkall\OneDrive - aueb.gr\MASTERS\THESIS\Github FIles\Code and Datasets> python main_one.py
? Please select a Folder:  (Use arrow keys)
 ❯ Datasets\golden-et-al-1998-set-1
   Datasets\li-et-al-2005
   Datasets\verolog-members-vrp-2016
? Please select a Folder:  (Use arrow keys)
 ❯ Datasets\golden-et-al-1998-set-1\Golden_01.xml
   Datasets\golden-et-al-1998-set-1\Golden_02.xml
   Datasets\golden-et-al-1998-set-1\Golden_03.xml
   Datasets\golden-et-al-1998-set-1\Golden_04.xml
====================================================================================================
Solving Datasets\golden-et-al-1998-set-1\Golden_01.xml...

  Hard-clustered VND:
  All nodes routed: True
  Nodes were routed only once
  No calculation error!
  Done!
  Optimised solution cost: 6348.6329

  Hard-clustered VND Multiple restarts, initial solutions with nearest neighbor rcl algorithm (rcl length = 3):
100%|██████████████████████████████████████████████████████████████████████████████████| 1/1 [00:00<00:00, 26.50it/s]
  Failed to improve after 500 restarts
  All nodes routed: True
  Nodes were routed only once
  No calculation error!
  Done!
  Optimised solution cost: 6348.6329

  Hard-clustered VNS
  Improving solution..
  Executed 32 times, improved 2 times, from 6348.6329 to 6319.1899
  All nodes routed: True
  Nodes were routed only once
  No calculation error!
  Done!
  Optimised solution cost: 6319.1899

----------------------------------------------------------------------------------------------------

  Soft-clustered VND:
  All nodes routed: True
  Nodes were routed only once
  No calculation error!
  Done!
  Optimised solution cost: 6354.9468

  Soft-clustered VND Multiple restarts, initial solutions with nearest neighbor rcl algorithm (rcl length = 3):
100%|██████████████████████████████████████████████████████████████████████████████████| 1/1 [00:00<00:00, 19.73it/s]
  Failed to improve after 500 restarts
  All nodes routed: True
  Nodes were routed only once
  No calculation error!
  Done!
  Optimised solution cost: 6354.9468

  Soft-clustered VNS
  Improving solution.......
  Executed 29 times, improved 7 times, from 6354.9468 to 6308.5987
  All nodes routed: True
  Nodes were routed only once
  No calculation error!
  Done!
  Optimised solution cost: 6308.5987

====================================================================================================
```

Else, if the user wants to avoid installing PyInquirer (as it is incompatible with Jupyter Lab and only one of the two can work at a time) he can follow the commented instructions in main_one.py or main_all.py. Note that in this way the user will have to assign manually the directory of the desired dataset.

2. Scripts main_*.py

The files main_one.py and main_all.py are used to:

1) solve one or more instances respectively by using the VND, VND MR, and GVNS methods (explained later) using soft and hard clustering
2) draw all the solutions, including the initial ones
3) save all results to the folder Plots_Results, in a sub-folder named after the runtime datetime.

The user can specify the maximum iterations and limit for the VND MR method, as well as the execution time for the GVNS method.

Both main files depend on SolverPackage to successfully run.

3. SolverPackage

3.1. Data.py

This file contains a single class called **Data**. The class is used to extract and organize data from an input file. The input file is expected to be in XML format and should contain information about nodes, requests, and vehicle profiles in a VRP-REP compliant format.

The class has the following methods:

1) **extract_data(self)**: This method is used to extract the data from the input file and organize it into dictionaries. It reads the input file using the **xml.etree.ElementTree** module and extracts the data for the 'nodes' and 'requests' sections. It then populates the class variable **self.nodes** with the extracted data. The method then extracts data for the 'vehicle_profile' section and populates the class variable **self.vehicle_profile**. Finally, the method returns the number of nodes, and the capacity of the vehicle.

2) **get_dem_stats(self)**: This method returns the statistics of the demand of the nodes. It first creates a DataFrame of the **self.nodes** dictionary using the **pandas** library, then finds the mean and std deviation of the demand of nodes and returns these values.

The class uses several functions to parse and access the XML tree, such as **findall()**, **find()**, **get()**, **text** and so on. These functions are provided by the xml library ElementTree.

The class also depends on the pandas library that is imported at the top of the file, used in the get_dem_stats method as well as to_numeric and errors='coerce' functions.

3.2. Clustering.py

This file contains a single class called **Clustering**. The class is used to create the clusters of nodes based on their positions, using the K-means algorithm from the **sklearn.cluster** module.

The class has the following methods:

1) **build_clusters(self, n_clusters)**: This method takes one parameter **n_clusters**, which represents the number of clusters to be created. It uses the **KMeans** class from the **sklearn.cluster** module to cluster the nodes and create the clusters. It takes nodes information and apply kmeans clustering algorithm on the positions of nodes, the number of clusters are **n_clusters**. Then it sets the cluster values for each nodes, and removes the last row (that represents the Depot) from the dataframe. It also sets the cluster for the Depot. Finally, it stores the centroids of the clusters in the **self.clusters** variable and returns whether the solution is solvable or not by checking the total demand of each cluster against the vehicle's capacity.

2) **create_clusters(self, initial_number)**: This method takes one parameter **initial_number**, which represents the initial number of clusters to be created. It repeatedly calls the **build_clusters()** method and increases the number of clusters until a solvable solution is found. It returns the number of clusters that resulted in a solvable solution.

The class uses several functions to parse and access the dataframes, all of which depend on pandas, numpy, and scikit-learn libraries. It is worth mentioning that the file uses a random seed so as to produce the same results on repeated runs.

3.3. Model.py

This file contains the classes that comprise the base structure of the entire approach: **Node**, **Route**, **Cluster**, **ClusterRoute**, and **Model**.

1) The **Node** class is used to create a node object that holds information about a node's ID, x, y position, demand, cluster, and its neighbors.
2) The **Route** class is used to create a route object that holds information about a route's sequence of nodes, the total cost, and the total load.
3) The **Cluster** class is used to create a cluster object that holds information about the cluster's ID, x, y position, nodes in the cluster, and total demand. It also has a method **calc_dem()** to calculate the demand of the cluster.
4) The **ClusterRoute** class is used to create a cluster route object that holds information about a cluster route's ID, sequence of clusters, the node route, the number of customers, the total cost, and the load.

The **Model** class is used to create a model of the CluVRP problem. It has several instance variables including the data and clustering objects, the number of clusters, the list of all nodes, the list of clusters, the list of customers, the distance matrix, and the cluster distance matrix.

It also has a **build_model()** method that initializes the model's variables and objects, and sets up the VRP problem by calculating the distance matrix of the nodes, and the distance matrix of the clusters.

3.4. Vrp.py

The Vrp.py file defines a class called **Vrp**, which contains methods for solving the Vehicle Routing Problem (VRP). The class takes a **Model** object as an input, which contains information about the clusters, the cluster distance matrix, the number of clusters, and the vehicle's capacity.

The main method of the class is **clarke_wright()** which uses the Clarke-Wright algorithm to solve the VRP. The algorithm is implemented in three steps:

1) Calculate the savings $s(i,j) = d(D,i) + d(D,j) - d(i,j)$ for every pair (i,j) of demand points. Where D is the depot and $d(i,j)$ is the distance between nodes i and j.
2) Sort the savings by descending order and keep only the positive savings.
3) Merge routes considering capacity and routing constraints. Starting with the highest saving, it iterates through all the routes and checks if it is possible to merge it with the route considering the capacity constraints. If the merge is feasible, it is performed, and the load of the route is updated.

Continuing, the method verifies that the cluster routes are correct and feasible and returns the cluster_routes list.

3.5. Tsp.py

This file contains an implementation of the Traveling Salesman Problem (TSP) algorithm. It takes as input the **Solver** object containing information about the nodes, clusters, and cluster routes, as well as configuration options such as the construction method and the length of the restricted candidate list (RCL). It uses the below methods:

1) The **initialize_tsp** function finds for each cluster route the nearest node to the depot, by looking at each node in the cluster and setting the nearest one as the "nearest_node" of the cluster_route.
2) The **solve_randomly** function takes a list of nodes and current route and randomly inserts them into the route.
3) The **find_neighbors** function finds a user-defined number of the nearest neighbors (RCL) of a node in a list of nodes and stores them in the node object so that it can be accessed later. For RCL = 1 it finds the one nearest neighbor
4) The **apply_nearest_neighbor** function applies nearest neighbor algorithm on the given node list, it constructs the route by picking the nearest unvisited node from the last node, by utilizing the find_neighbors function.

5) The **construct_solution** is the main function to be called by the user to solve the TSP problem. For hard methods it applies the nearest neighbor strictly to each cluster in each cluster route, solving in that way many open-type TSPs. For soft methods it applies the algorithm on the whole cluster route, solving in that way a closed TSP. Finally it updates

3.6. Optimization.py

This file contains the base structure of the optimization methodologies of this implementation.

It contains three move type objects:

1) The **RelocationMove** class has three attributes: original_position, new_position and cost. The initialize method sets the cost to a large value and the original and new positions to None.
2) The **SwapMove** class also has three attributes: first, second and cost. The initialize method sets the cost to a large value and the first and second positions to None.
3) The **TwoOptMove** class has the same attributes and initialize method as the SwapMove class.

The **Optimisation** class is the main class of the file. Its methods aim to optimize the initial solution, and are listed below:

1) The **create_subroutes** method, in case of using hard clustering constraints, takes a sequence of nodes and splits it into subroutes for each distinct cluster.
2) The **calculate_route_cost** method takes a sequence of nodes and returns the total cost of the route.
3) The **find_best_relocation_move**, **find_best_swap_move**, and **find_best_two_opt_move** methods find the most profitable relocation, swap and two opt move respectively for a given sequence of nodes, and store this move's characteristics (cost, nodes that need moving) in the respective move type object
4) The **apply_relocation_move**, **apply_swap_move**, and **apply_two_opt_move** static methods take as input a sequence of nodes and a move type object, and apply the stored move to the sequence.
5) The **local_search** method takes a sequence of nodes and iterates through the list, repeatedly finding the most profitable move possible (depending on the move type) and applying it to the route, until no further improvement can be made.
6) The **vnd** method does exactly the same as the local search, but when it stops finding improvements with a move type, it changes the operator to further explore the solution space. It starts with the relocation move type, then moves to swap, and finally two opt. It then cycles through these move-types repeatedly. If using all three operators does not reduce the route cost, the method terminates and returns the best route it found along with its cost.

7) The **optimize_route** method takes a sequence of nodes as input and optimizes it either with the **local_search** or with the **vnd** method.
8) The **optimize_solution** is the main method of this class, takes as input a **Solution** object, and uses all the above methods to optimize it.
9) The **randomize_route** method takes a sequence of nodes and relocates a predetermined number of nodes randomly, depending on the length of the sequence. Specifically, a third of the total number of nodes in the sequence.
10) The **randomly_select_neighboring_solution** method takes a **Solution** object and randomly selects a neigboring solution by using the **randomize_route** method on the solution's routes.

3.7. Solver.py

Solver.py contains two classes, the **Solution**, and the **Solver** class.

The **Solution** class contains the necessary information about the solution of the problem, specifically:

1) Total cost
2) Cost per route
3) Routes

It also has three methods:

1) **back_up** which is meant to duplicate the Solution object
2) **save_to_txt** which saves the solution to a txt file named "results"
3) **__str__** which is used to print the solution details in a human readable format

The **Solver** class takes a **Model** object as input, and combines the classes and methods of Vrp.py, Tsp.py, and Optimization.py to solve a single instance and verify the validity of the solution.

Its methods are listed below:

1) **reset** is meant to reinitialize all the necessary variables of the problem
2) **check_solution** takes a **Solution** object as input and checks its validity in the following aspects:
    a. all nodes are routed
    b. capacity constraints are not violated
    c. each node is visited once
    d. the solution's costs were correctly calculated
3) **solve** method is used to solve a single instance of CluVRP. The user can specify whether the problem will be solved as hard or soft clustered (hard = True/False), the construction method of the initial solution (nearest

neighbor = 0, random=1), the length of the restricted candidate list for the nearest neighbor method (length=1 to not randomize the results of nearest neighbor algorithm), the optimization method (local search = 0, VND = 1), and finally the move type of the local search method (relocation=0, swap=1, two opt=2)

4) **vnd** method uses the **solve** method preconfigured to run with vnd. The user can specify the construction method of the initial solution, whether the problem will be solved as hard or soft clustered, and whether to print the details of the solution or not.
5) **vnd_mr** method uses the **vnd** method with multiple restarts to further explore the solution space. The user can define the maximum iterations, the limit of unprofitable iterations to terminate earlier the algorithm and the rest of the **vnd** method's parameters.
6) Finally, the **vns** method uses the **vnd** method to obtain a predetermined good solution, and then utilizes the **randomly_select_neighboring_solution** method of the **Optimization** object to randomly obtain a neighboring solution. It then uses the **vnd** method again to optimize the neighboring solution. If the optimized neighboring solution is better than the previous best solution, it becomes the new best solution and the algorithm repeats these steps for a user-defined time. The user can also define the rest of the **vnd** method's parameters.

3.8. SolDrawer.py

This file contains a single class named **SolDrawer** which uses its methods to draw and save the solutions' figures.

3.9. main_functions.py

The functions in this file combine all the above .py files to solve a single instance of the problem with **vnd**, **vnd_mr**, and **vns** methods, verify the solutions, draw the solutions' figures, and organize the results of each run based on date and time.

The functions **select_file**, **get_instance_name**, **solve_and_draw**, **create_results_dictionary**, and **save_results** are necessary to execute the main scripts.