MÁSTER UNIVERSITARIO EN INVESTIGACIÓN EN INTELIGENCIA ARTIFICIAL

Resolución de problemas con metaheurísticos - Curso 2019/20

Trabajo 2.10. Documentación del VNS.

Realizado por:
Guillermo Tomás Fernández Martín
fernandezmartin@posgrado.uimp.es

Albacete, 25/02/2020

Contents

1 Introduction **1**

2 Language choice **1**

3 Data Structures **1**

3.1 `coord_map` 1

3.2 `routes` 2

4 Functions **2**

4.1 `distance` 2

4.2 `route_length` 2

4.3 `validate_route` 3

4.4 `augerat_parser` 3

4.5 `routes_score` 3

4.6 `intra_swap`, `intra_shift`, `inter_swap` and `inter_shift` . . 3

4.7 `sequence_exchange` 3

4.8 `VND` and `VND_movement` 4

4.9 `MC2` 4

4.10 `SE_MOVEMENT`, `SE2` and `SE3` 4

4.11 `shake` 5

4.12 `build_routes` and `build_initial_solution` 5

4.13 `general_VNS` 5

4.14 `general_VNS_small`, `general_VNS_mid` and `general_VNS_big` 6

5 Execution and experiments **6**

1 Introduction

Along this document, the VNS implemented to solve a variation of the CVRP problem will be explained. This document will cover the data structures, functions and design choices taken during the development of the algorithm.

2 Language choice

For the implementation of the algorithm, Python has been the language used. The choice has been motivated by several factors. On the one hand, it simplifies some of the mathematical operations to be made, given that they are already implemented in the core libraries. On the other hand, it simplifies the implementation of the methods, given a cleaner, simpler syntax compared to other languages such as Java. Finally, it allows for some functionalities, such as using functions as parameters; that allow a further abstraction in some parts of the code. It is noticeable, however, the downside of this choice. Given that Python is an interpreted language, the code will run far slower than in C or Java. For the scope of this subject, however, it has been decided that this factor is not important enough, so this language will be the final choice.

3 Data Structures

There are two general structures that will be used along most functions of the algorithm: `coord_map` and `routes`.

3.1 `coord_map`

This structure is the one that contains the map of the place with the coordinates of the nodes. It is a dictionary, given that is one of the most efficient structures in Python for quick data access. The key of the dictionary is the index of the node, and the value is a second dictionary with both the x and y values. This is a very efficient way of accessing the values, as well as a self-explained structure for any later modification of the code. It is simpler to understand what `node['y']` means than `node[1]` when revisiting the code (if it were a tuple instead of a dictionary), and it is as efficient. In the same way, it is clearer to return the key of the node to know the order of the routes

in the solution after shuffling and changing the order of the stops to generate some of the solutions.

3.2 `routes`

This structure will be the final objective of the algorithm. It is a list of dictionaries, where each dictionary contains the route of a determined truck. In this case, each route is a dictionary to allow for expandability. At the beginning of the essay, every truck was to have a different capacity, which would also be allocated in this structure. However, later on it was decided for all the trucks to share the same capacity to keep the problem simple. The structure was kept a dictionary to keep the code functional and expandable, given that the performance would not vary significantly from using simple lists. The dictionary, then, is composed by a truck and a list of stops that the truck will traverse. This list is considered to be ordered for the final solution, and it does not include the depot. This way, a route with the stops `[3,7,2]` will depart from the depot, go to the node `3`, then the node `7`, the node `2` and return to the depot.

4 Functions

In this section, the different functions that make the code work will be explained. In this document, the particularities of the algorithm will not be explained, but rather the implementation decissions.

4.1 `distance`

Auxiliary function that computes the Euclidean distance. It takes advantage of the `math` library, using its functions to get the distance.

4.2 `route_length`

Auxiliary function that computes the distance of a whole route. It computes the distance between all stops, plus the distance from the depot to the first node and from the last node to the depot (that are not specified in the route). If a route is empty, it returns 0, so there is the possibility of

eliminating trucks if it is beneficial for the solution.

4.3 `validate_route`

Auxiliary function that checks the restriction of the maximum length a truck can traverse. This behaviour has been extracted to a function for clarity in the code.

4.4 `augerat_parser`

Auxiliary function that takes a file in the **Augerat** format and extracts the information of the nodes to create the structure `coord_map`

4.5 `routes_score`

Auxiliary function that computes the score of a solution. Again, this has been extracted to a function for clarity and reusability in the code.

4.6 `intra_swap`, `intra_shift`, `inter_swap` and `inter_shift`

Functions that control the movement of the stops of the routes as defined in the algorithm. They take the route or routes and the indices of the elements, and they perform the movement, returning a new route. They have to be called explicitly with the indices, so that it can be used in both random movements and exhaustive movements (shake or VND). They generate a copy of the original structure, modify it and return it, so that if this element is discarded (for example, a low score) the original element is not modified.

4.7 `sequence_exchange`

Function that exchanges a sequence of a determined length between two routes. It uses internally the `inter_swap` movement as many times as the sequence length. It acts as a wrapper to change neighbourhoods in other parts of the algorithm.

4.8 `VND` and `VND_movement`

These two functions control the local search section of the algorithm. The `VND` function keeps on calling the `VND_movement` function for as long as there is an improvement in the new route this one returns. The `VND_movement` function makes an exhaustive search of the movements available until it finds a route that improves or finishes. First, it tackles the movements that only involve a single route (`intra_movements`), and then the ones that involve two routes (`inter_movements`). However, the function shuffles the order of these group of movements, so that it adds a layer of uncertainty.

Given that the `VND` is where the difference between the algorithms reside, both functions take a list of movements that can be performed. This list is composed by functions, so that it can be shuffled, iterated and accessed independently of the number of functions or actual function in the list (as long as it has the same parameters in the same place). This is also why `VND_movement` is divided between `inter` and `intra` movements: as they act over a different number of routes and have a different parameter layout, they have to be separated.

4.9 `MC2`

Function that takes care of the *Move Combination* neighbourhood for the `shake` function. This function takes care of choosing a random movement between the four available, choosing random, different, valid routes, executing the movement and returning it if it generates a valid route. This means that if the movement needs two routes they cannot be the same, or if it only needs one route this needs to have at least two elements. Also, takes care of removing the executed movement from the list, so that the same movement is not executed twice.

4.10 `SE_MOVEMENT`, `SE2` and `SE3`

This functions take care of the *Sequence exchange* neighbourhoods for the `shake` function. `SE_MOVEMENT` executes a sequence exchange movement of a determined length. It takes care of choosing two different routes that are eligible for this movement (if the route is only two movements long, it cannot execute a three stop sequence exchange, for example). Also, takes care of choosing a correct index (not choosing the last element of a route as the start of the sequence exchange). Finally, it also checks that the generated route

is valid. Both `SE2` and `SE3` functions act as wrappers. They have the same parameters as `MC2`, so it is possible to abstract from this functions inserting them in a list, in the same way as with the movements.

4.11 `shake`

This function takes a number k of neighbourhood changes and picks k random movements from `MC2`, `SE2` and `SE3`. These movements can be called once or twice, so that there are six different possible neighbourhood changes. As the validity of the new routes is checked in each individual movements, this function gets very simplified.

4.12 `build_routes` and `build_initial_solution`

These functions take care of the greedy algorithm that builds the initial solution. The method `build_initial_solution` starts with a single truck and generates a solution. If it is not valid, it increases one truck and tries again. This generation is done by the `build_routes` function. It generates a random order of all the nodes but the depot, and tries to add each stop to every route. Then it adds it to the shortest new route, this is, the route that is the shortest with this new stop. This generates an initial solution that has a random component, so that every iteration of the algorithm changes; but is not completely random, but oriented to get a good solution, which improves the chance of the VNS getting a better solution.

4.13 `general_VNS`

This function executes the algorithm using all the functions already explained. It takes the path of the configuration file, the maximum k for the neighbourhood change, the maximum capacity of the trucks and the lists of movements that can be performed. It can also change the verbose level for debugging purposes. It is a straight up implementation of the algorithm: while the neighbourhood change is not greater than the makimum, it shakes and improves the best solution. If the new solution improves the best so far, it resets the neighbourhood change, else it increments the number of times it will change. Finally, it returns the best score obtained, and depending on the verbose level also prints the routes obtained.

4.14 `general_VNS_small`, `general_VNS_mid` and `general_VNS_big`

Finally, these functions act as wrappers of the `general_VNS` method with the different number of movements. These are the functions that will be exported and called to generate the results.

5 Execution and experiments

For the execution of the algorithm, a different file has been created. This new file imports the wrapper VNS functions to call them. It parses the filename passed via command line and calls 30 times the three variants of the VNS with different seeds. After executing the 30 iterations, it prints the mean, median and best execution and writes the result in a file. To perform a statistical analysis, there is a wrapper function that checks the folder where all the result files are written and merges them in a single file.

To call the `A-n32-k5` experiment function, use:

```
python vns_experiments.py instances\A-n32-k5.vrp
```

The function parses the backslash (\) character as separator because it was programmed for a Windows environment originally.