**Heuristic Techniques**

**N: number of cities**

**K: number of deliverymen**

**Minimize: f1 = The total length of the tour**

**: f2 = The longest path of deliverymen**

We need a constant **w1** in the range [0, 1] representing to what degree each of the functions impacts the objective function value.

Denote f is a **weight function**

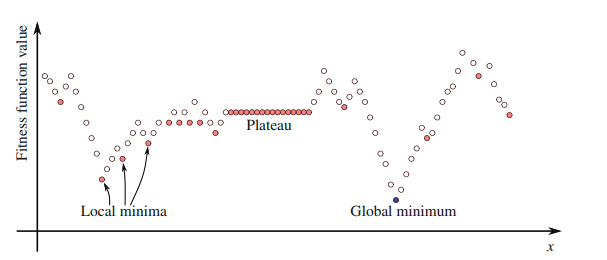
**-> Minimize: f = w1 \* f1 + w2 \* f2 ( w2 = 1 – w1)**

I will choose **w1 = 0.0005** because we want to optimize f2 first. If there is no improvement of f2, we then optimize f1. If no such improvement is found on both f1 and f2, it stops.

***Note:***

Local Optima:

* There is no guarantee that the returned solution is the best for the particular instance.
* A plateau is a set of neighboring solutions all having the same value.
* Objective functions such as min(max(…)) generate many plateaus with many solutions. Their optimization with a local search is therefore difficult.



If we try to optimize only objective function f2 which is min(max(…)) with local search. It generates many plateaus and we will obtain an insufficiently “good” solution. To avoid it, I try to use weight function f. You can use some Metaheuristic techniques such as: Simulated Annealing, Tabu Search,…

**I. Local Search**

**Sourcecode: LocalSearchVRP.py**

The general idea is therefore to start from a solution obtained by randomizing and improve it locally. The process is repeated until no further improvement is achieved.

**Algorithm**: General framework of local improvement method.

**Input:** Solution ***s***, method modifying a solution

**Result:** Improved solution s

1 **repeat**

2 **if** *there is a modification of s into s’ improving s* **then**

3 *s ← s’*

4 **until** *no improvement of s is found*

We need a method to modify the solution or generate a neighborhood set N(s) of a solution s. I will present 4 operators to do this: 2-opt Intra-Route move, Swap, Relocate, Swap, 2-opt Inter-Route move.

**Implementation:** The general idea is to use 4 operators to try to improve the solution quality by considering 4 potential moves. Choose the move that gains the best improvement (decreases the objective function most) and perform it. Each operator will provide a potential move according to First Improvement policy. I will go into more detail about each operator later. If no improvement is found, the process stops. Here is the framework of Hill-Climbing

General Framework of Algorithm I:

**Input:** solution ***s***, function ***gain(…)*** to compute the gain of each operator and choose the best one.

**Output:** improved solution ***s***

1 **repeat**

2 **if** **gain(**operator1, operator2, operator3, operator4**)**<0 **then**

3 perform move that gains the best improvement (decreases the objective function most)

4 **until** **gain(**operator1, operator2, operator3, operator4**)** >= 0

**1. Operators**

Intra-route: a change within one route

Inter-route: involving at least two routes

**1.1 2-opt Intra-Route move**

This operator developed by Lin for the TSP can also be used for the Vehicle Routing Problem. It performs an exchange of 2 edges. Choose nodes ***i*** and ***j*** from the same route, with **i** + **1** < **j**. Delete arcs **i** -> **i** + **1** and **j** -> **j** + **1**. Replace with **i** -> **j** and **i** + **1** -> **j** + **1**. This reverses the order of nodes from **i** + **1** to **j**. Neighborhood size is . See **Figure 1.1**

Diagram

Description automatically generated

**Figure 1.1**: 2-opt intra-route

Implementation:

* 2-opt intra-route inputs are the maximal sequence length to exchange and a feasible solution. The maximal length of 2-opt can be configured by the user. 2-opt tries, starting with the first route, to perform an exchange and compute the gain (delta) of maximal length sequence. The move is only accepted if the new solution after performing move is feasible and the value objective function was reduced. If no such improvement is found on the first route, the length of the sequence to be exchanged will be decreased by one and again all exchanges on the first route will be examined. If no improvement is found with this operator with a length of two, then 2-opt continues by trying to examine the second route by exchanging sequences with maximal length. Of course, sequence length will be reduced if no improvement is found. If once a feasible improvement was found (First Improvement policy), then return the delta (gain) and potential move to Algorithm I to compare, then stop the process. It also stops after examining all exchanges with a sequence length less than the maximal length on all routes, without finding any improvement

**1.2 Swap**

It swaps city ***i*** and city ***j*** belonging to different routes (k1 and k2). See **Figure** **1.2**

A picture containing diagram

Description automatically generated

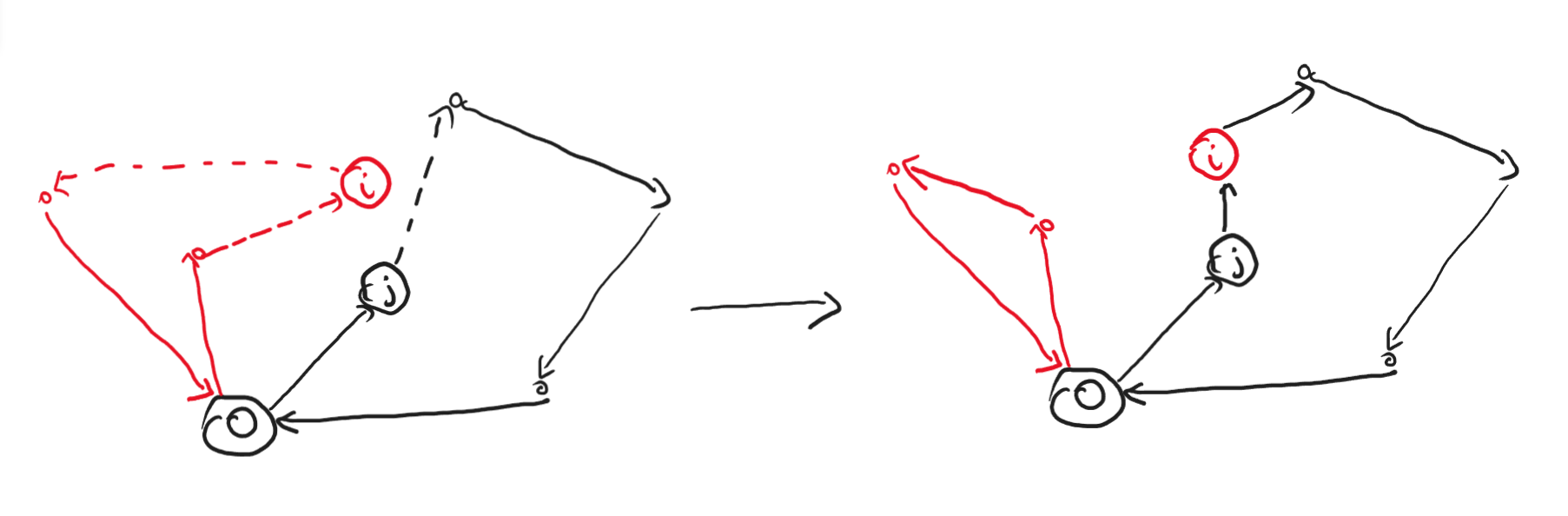
**Figure 1.2: Swap**

Implementation:

* Swap input is a feasible solution. Swap tries, starting with the first route, take the city (city i) and the second route, also take the city (city j) to perform an exchange and compute the gain (delta). The move is only accepted if the new solution is feasible and the value of the objective function was reduced. If no such improvement is found on the first route and the second route, the k2 route will be changed to the third route and all exchanges will be examined. If no improvement is found with the first route and the last route, then the first route k1 will be changed to the second route and of course, all exchanges will be examined. The process will stop if two routes that need to examine reach the last route, without finding any improvement. If once a feasible improvement was found (First Improvement policy), then return the delta (gain) and a potential move to Algorithm I to compare, then stop the process.

**1.3 Relocate**

It takes city ***i*** from route k1and optimally insert it after city ***j*** into tour k2. See **Figure 1.3**



**Figure 1.3: Relocate**

Implementation:

* Relocate input is a feasible solution. Relocate tries, starting with the first route, take the city (city i), and the second route, also take the city (city j) to perform an exchange and compute the gain (delta). The move is only accepted if the new solution is feasible and the value of the objective function was reduced. If no such improvement is found on the first route and the second route, the route k2 will be changed to the third route and all exchanges will be examined. If no improvement is found with the first route and the last route, then the first route k1 will be changed to the second route and of course, all exchanges will be examined. The process will stop if two routes that need to examine reach the last route, without finding any improvement. If once a feasible improvement was found (First Improvement policy), then return the delta (gain) and a potential move to Algorithm I to compare, then stop the process.

**1.4 2-opt inter-route move (tails swap)**

It exchanges the tail of a route k1 (start from city i) and the tail of another route k2 (start from city j). See **Figure 1.4**

Diagram

Description automatically generated

**Figure 1.4: Tails swap**

Implementation:

* Tails swap input is a feasible solution. 2-opt inter-route tries, starting with the first route, take the city (city i), and the second route, also take the city (city j) to perform an exchange and compute the gain (delta). The move is only accepted if the new solution is feasible and the value of the objective function was reduced. If no such improvement is found on the first route and the second route, the route k2 will be changed to the third route and all exchanges will be examined. If no improvement is found with the first route and the last route, then the first route k1 will be changed to the second route and of course, all exchanges will be examined. The process will stop if two routes that need to examine reach the last route, without finding any improvement. If once a feasible improvement was found (First Improvement policy), then return the delta (gain) and a potential move to Algorithm I to compare, then stop the process. Note that we don’t need to examine the case that ***i*** is the first city of k1 and ***j*** is the first city of k2 or ***i*** is the last city of k1 and ***j*** is the last city of k2.

All these operators generate the neighborhood size of .

Other operators: 3-opt, Or-opt, Cross, Ejection chains, …

**II. Greedy Heuristic Based on Local Search for TSP**

Sourcecode: greedyVRP.py

Function: greedy\_optimalTSPtoVRP()

If we assume that there is one deliveryman (K=1), the problem becomes TSP (Travelling Salesman Problem). The general idea is therefore to start from a TSP solution obtained by randomizing and improve it locally by using some simple local search algorithms such as 2-opt, 3-opt, Ejection Chains,... The process is repeated until no further improvement is achieved. Now, we have the optimal solution for TSP. Back to the main problem, the VRP, we need to minimize the longest path of the tour, which means that the total length of the tour should be divided “equally” for everyone. It turns out that we can transform optimal solution for TSP into solution for VRP by using greedy algorithm. The general idea of greedy algorithm is to ”insert” sequentially the city in solution for TSP to the deliveryman until the length of this route approximates the average length which is equal to the total length of the tour divided by the number of deliverymen. If the length of the current route is larger than the average length, stop inserting the next city to this route and add the next city in the TSP solution to the next route. This method seems appropriate for Large Scale VRP (N > 200).

If you want to improve the solution after using greedy algorithm, you can generate the initial solution by using greedy algorithm and using technique that is mentioned in **I** (function **weight\_improvement\_heuristics**()). Because the technique in **I** is very slow for large scale, so I set the time limit for 200 seconds to exit from process and return the solution.

**1. TSP 2-Opt**

**Cite from [1]**

How to perform a 2-opt move in constant time?

* A data structure, inspired by the work of [2], enables performing a 2-opt move in constant time.
* For each city, an array stores both adjacent cities. The array components at indices ***2i*** and ***2i + 1*** provide both adjacent cities of the city ***i***
* An array ***t*** with ***2n*** indices represents a solution. Initially, ***t[2i]/2*** provides the number of the city succeeding ***i*** and ***(t[2i+1]−1)/2*** the number of the city preceding i. A 2-opt move consists in modifying four values of the array ***t***. This can be realized in constant time. **Figure 2.1** illustrates the operating principle of this data structure.
* This code below initializes an array t (using in **Fig. 2.1**) implementing this data structure from a given tour. The last is provided by the list of cities successively visited (and not the successor of each city).

Text

Description automatically generated

* **Code 2.2** implements a first improvement heuristic based on this principle. It uses the shift operator i >> 1 to quickly evaluate the expression i/2. The last is the number of the ith city. The “exclusive or” operator iˆ1 allows quickly calculating the expression i+1-2\*(i%2) providing the index to access the adjacent city.

Diagram

Description automatically generated

**Fig. 2.1** Data structure for performing 2-opt moves in constant time. The ***2i*** and ***2i*** + 1 entries of an array allow identifying both cities adjacent to the city ***i***. Starting from the index 0, we can reconstitute the tour by following the adjacent city. Starting from the index 1, we can reconstitute the tour in the other direction. Performing a 2-opt move consists in altering four entries in the array

**Code 2.2** Implementation of a first improvement heuristic for the TSP based on 2-opt neighborhood. This implementation exploits the data structure shown in Fig. 2.1 for performing the moves in constant time

Text

Description automatically generated

Other algorithms: 3-Opt, Or-Opt, Ejection Chains, POPMUSIC,…

**References**

1.É. D. Taillard, Design of Heuristic Algorithms for Hard Optimization, Graduate Texts in Operations Research, <https://doi.org/10.1007/978-3-031-13714-3_6>

2. Osterman, C., Rego, C.: A k-level data structure for large-scale traveling salesman problems. Ann. Oper. Res. 244(2), 1–19 (2016). <https://doi.org/10.1007/s10479-016-2159-7>

3. Michael Huemer, 2011, Heuristics for the vehicle routing problem with multiple deliverymen, München, GRIN Verlag, <https://www.grin.com/document/232841>

4. Editors: F. Rossi, P. Van Beek, T. Walsh, 2006, Handbook of Constraint Programming (Foundations of Artificial Intelligence), <https://www.sciencedirect.com/bookseries/foundations-of-artificial-intelligence/vol/2/>