LIFE GreenYourRoute: A European innovative logistic platform for last mile delivery of goods in urban environment

Deliverable C3.1: *Report focusing on the results achieved and/or deviations experienced/expected as compared to the original estimates/inputs in the KPI Webtool and LIFE performance Indicators*

Simulation Tool

Partner responsible for this report: UTH

Document Information Summary	
Action:	**C3 Update and Monitoring of Key Project-level Indicators**
Sub-action:	**Sub-action C3.1: Update of Key Project-level Indicators** **Sub-action C3.2: Monitor and evaluation Key Project-level Indicators**
Deliverable Number:	**C3.1**
Deliverable Title:	**Report focusing on the results achieved and/or deviations experienced/expected as compared to the original estimates/inputs in the KPI Webtool and LIFE performance Indicators**
Leader:	**UTH**
Participants:	**-**
Author(s)	**Dr. Georgios K.D. Saharidis, Zoi Moza, Georgios Kalantzis, Neilos Psathas.**
Project website	**www.greenyourroute.com**
Status:	**Final**

Disclaimer:

The LIFE GYR [LIFE17 ENV/GR/000215] project is co-funded by the LIFE programme, the EU financial instrument for the environment.

The sole responsibility for the content of this report lies with the authors. It does not necessarily reflect the opinion of the European Union. Neither the EASME nor the European Commission are responsible for any use that may be made of the information contained therein.

Start Date: 01 September 2018 - Duration: 42 months

Contents

Contents 4
1 Introduction 5
2 Algorithm implementation 5
2.1 Step 1: Calculation of weighted center 5
2.2 Step 2: Statistical analysis 6
2.3 Step 3: Scanning 7
2.4 Step 4: Problem re-assignment of customers to salesmen 9
2.5 Step 5: Re-assignment of customers to a different salesman. 10
2.6 Step 6: Optimal routing plan of trucks 13
3 Conclusion 14
4 Annex 15

1 Introduction

In this deliverable, the developed by UTH simulation tool in order to monitor the environmental and socio-economic impact of the project by performing a comparison between the former empirical routing planning approach and the routing planning approach resulted by GYR service is presented.

As the demonstrators will stop following empirical approaches to create, their daily routing planning, due to the use of GYR service, a simulation tool to define these potential routing plans (i.e. empirical plans) should be developed.

The simulation tool used the input data (i.e. daily orders, available fleet of vehicles etc.) of the demonstrators into the GYR platform in order to generate the potential routing plans in case an empirical approach was followed.

This simulation tool is a heuristic routing tool which simulates the former approach providing sub optimal and feasible routing plans based on daily routing requests. The simulation tool includes emission calculation models used in GYR platform and heuristic rules such as heuristic clustering approach grouping customers based on geo-data, heuristically decomposing the problem into vehicle selection and then assignment of customers to the selected vehicles, defining visiting order of customers based on the criteria of the nearest neighbor. This simulation tool provided us with the opportunity to obtain the empirical solution fast with an automatic, systematic and easy way, without the demonstrators having to invest effort and spend time (i.e. minimum 5 hours per day) to provide a solution according to the former approach.

The Simulation tool includes an optimization problem where a number of trucks need to visit a number of customers in order to deliver products or collects orders or pick up products.

2 Algorithm implementation

The ultimate goal of the algorithm is to give each demonstration a daily routing plan. The steps taken to solve this problem are explained below:

2.1 Step 1: Calculation of weighted center

The first step is to calculate the weighted center of customer points. By reading the input data and knowing the geographic coordinates of each customer, it is trivial. It is noted that points are taken as points in the Cartesian plane where the customer coordinates are translated into Cartesian coordinates. Calculation of the weighted center is done by computing:

$$\bar{x} = \frac{1}{N}\sum_{i} x_i, \bar{y} = \frac{1}{N}\sum_{i} y_i$$

where N is the total number of customers ανδ (x_i, y_i) are the coordinates. The following figure present graphically the weighted center of the reference example.

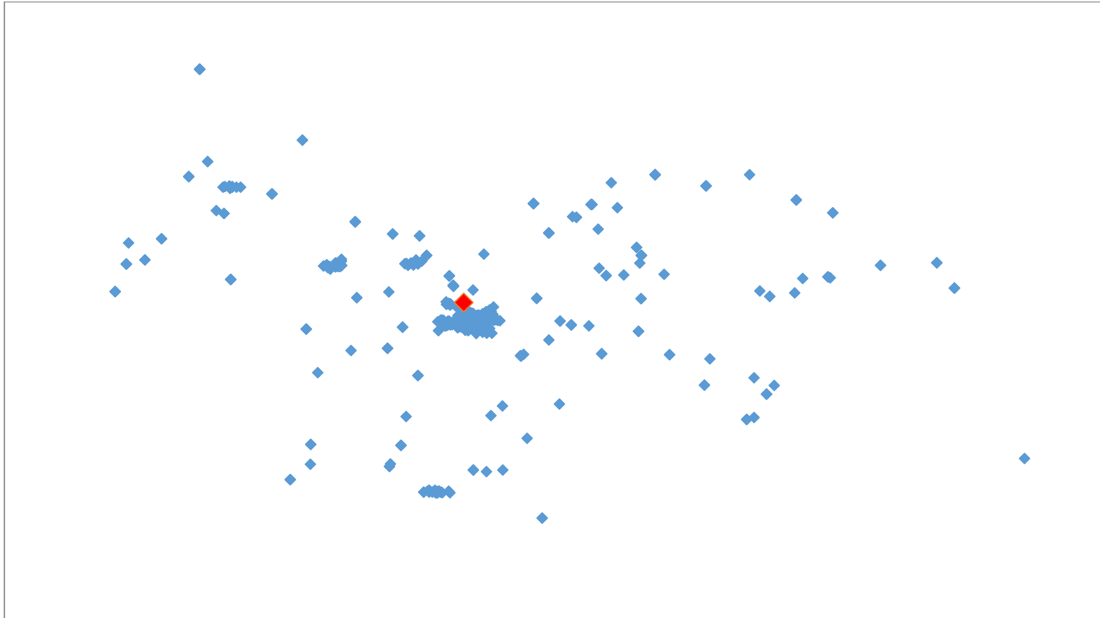

Figure 1: Weighted center of reference example

This step simulate the first action of the person responsible to create the daily routing plan referred to as routing planner. This step is not performed in a daily base by the routing planner of ATHINAKI, PLUS and KOUKOUZELIS. This step is performed in a daily base from the routing planner of the customer of CEDA and ITACA.

2.2 Step 2: Statistical analysis

In the second step, the algorithm starts with a brief statistical analysis of the data that corresponds to customers. Returns user information about the problem that may help him identify errors or noise that may have occurred in the data. For the volume and distances from the center of gravity, the algorithm calculates mean value and variance. It returns to the user all clients who have extreme behaviors, i.e. they have an order (i.e. volume) in excess of $\mu + 2\sigma$ (where the average and σ the standard deviation of the population). The statistical analysis checks whether the population has a high coefficient of variation (σ / μ) and informs the user about it.

An example provided by the simulation tool is the following:

```
While the average turnover is 206.13 (but highly volatile) the following customers have
extremely higher turnover
Customer ID: 111 Volume: 4573.96
Customer ID: 166 Volume: 1709.84
Customer ID: 423 Volume: 1408.43
Customer ID: 167 Volume: 1233.82
Customer ID: 537 Volume: 1229.86
Customer ID: 604 Volume: 1139.54
Customer ID: 105 Volume: 1131.54
……………………………………………………………
The following customers are far away from the centre of all points
Customer ID: 15
```

Customer ID: 16
Customer ID: 47
Customer ID: 51
Customer ID: 59
Customer ID: 262
Customer ID: 280
Customer ID: 281
……………………………………………………

Figure 2: Statistical Analysis of the data

This step simulates the customer screening in order to identify if they are customers with order of high volume and a special truck should be used to satisfied them.

2.3 Step 3: Scanning

The goal in Step 3 is to create original customer clusters with a single objective, in this algorithm step, the quantification of a number of visits to each truck. We choose a random customer of the grid and create a line that starts from the center of gravity (i.e. weighted center) of the cloud of visiting points and extends in the direction of the random customer called the "starting point". So, we create the following line in the reference example:

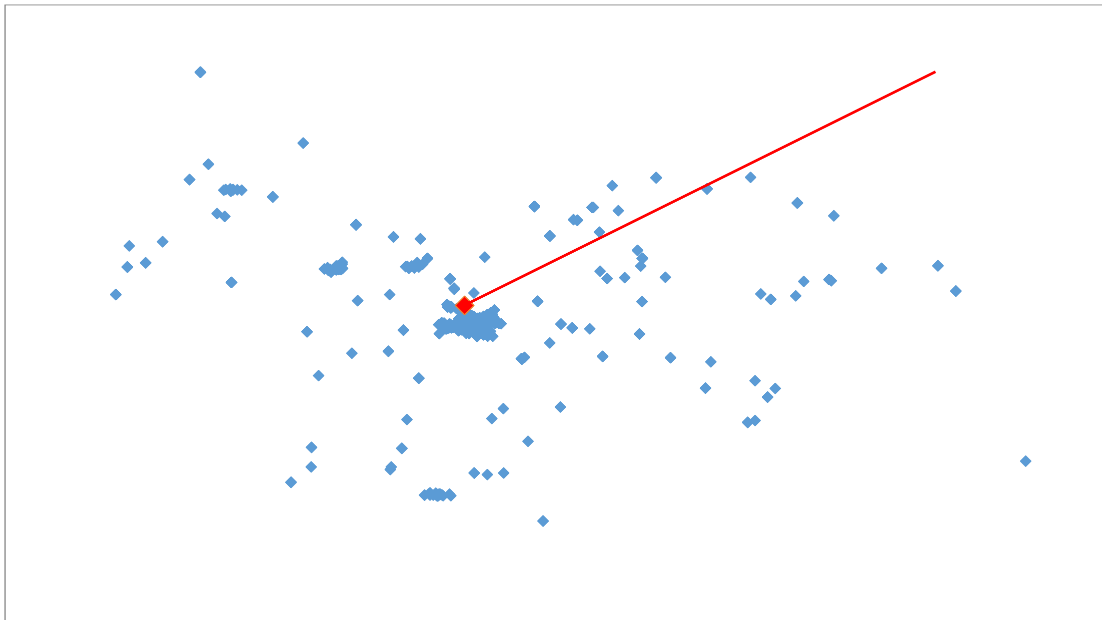

Figure 3: Scanning reference example data

We consider this line as the reference axis, which for each customer defines an angle φ in relation to it. Apparently the original customer that corresponds to the "starting point" has φ = 0 while all other customers are sorted in ascending order based on their angle φ_i. In defining the area of responsibility for the first vendor, we sense the reference axis from the position to which it is clockwise assigning clients to the current area of responsibility until the number of visits by all the clients in the area is equal to the average number of visits we would like every specific truck has. As soon as the number of customer visits in the current responsibility area, that is to say, between the initial and final radius of the scanner we created, is probably the average number of visits, we stop creating the current responsibility area, defining the area of

responsibility of the first vendor. The above procedure is done as many times as the number of available trucks in order to define for each of them their area of responsibility. Applying this procedure to the reference example, the following three areas (i.e. assuming 3 trucks are available for this urban region) of responsibility are schematically created:

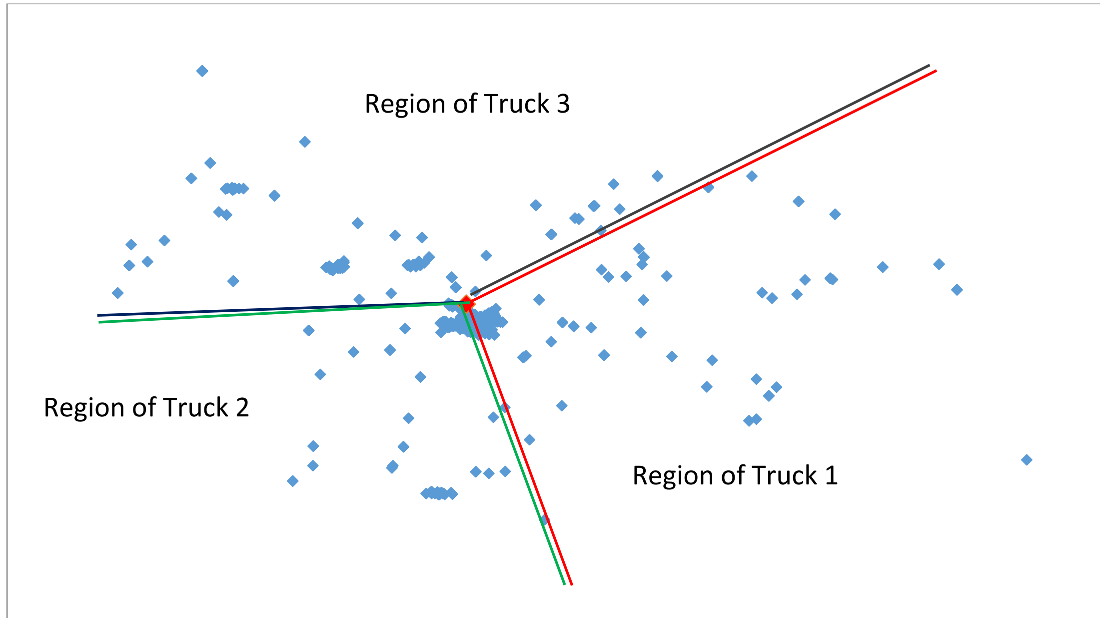

Figure 4: Region of responsibility for each Truck.

The algorithm could be enough for the original random customer to define the reference axis, and the process of assigning the responsibility areas to be completed once. For the better creation of responsibility areas, the algorithm repeats the above steps with different original customers by taking different areas of responsibility for each truck. The best result of areas of responsibility is defined as the result where the areas of responsibility share as far as possible the distant points between them. In a good result of liability areas vendors will be as good as remote customers so that the workload for each of them is equalized.

The following simple procedure is used to evaluate each result of liability areas. Each responsibility area has its own center of gravity and a distance from the center of gravity of all points as calculated above in Step 1 of the algorithm. Another cluster has a remote center of gravity in relation to the whole, the other closest, thus defining how remote the truck's area of responsibility is. To find the best solution of liability areas we check for each solution anymore is the distance each center of gravity of each seller from the center of gravity of all points. In the end, we choose the solution whose distance between the most distant from the center and the closest to the center area of responsibility is minimal. Note that there may be alternative ways of implementing this step, such as examining the combination of liability areas with the smallest fluctuation in the distances between their weight centers and the overall cloud center of gravity of the cloud.

At the end of this step, initial areas of responsibility have been defined based solely on the number of visiting points. It is possible, due to the data of the problem to be solved, whether the customer orders are added to each area of responsibility and the orders volumes are not

evenly distributed to trucks as we would like. To this end, the next Step 4 was developed to resolve any such problems.

This step simulates the closeting approach of the routing planners who selects visiting points to be assigned to a truck.

2.4 Step 4: Problem re-assignment of customers to salesmen

Previous Step 3 gave us an initial customer assignment to trucks, but it may be problematic in terms of the number of orders (i.e. volume) that each truck transports.

To solve this problem, we developed a model of mixed integer mathematical programming that aims to transfer customers from one truck to another truck in order to achieve the best possible volume for each truck.

We declare that the cost of assigning the customer i to the truck k is 0 if the customer is ultimately assigned to the truck initially assigned to the previous Step 3. If given to any other truck, then the cost is 1.

The following definitions of indexes and data are used:

Indexes	Description
i	Customers
k	truck
vV	Range of fluctuation of the total number of visits
vT	Range of fluctuation of orders/volume
Parameter	**Description**
q_i	Order/volume of customer i
$\bar{Q}$	Average level of orders
$\bar{V}$	Average number of visits
f_i	Frequency of visits for customer i
c_{ik}	Cost to assignment customer i to salesman k, 0 if the customer stay assigned to the initially assigned salesman in step 3 and 1 if the customer is re-assigned to a different salesman.

Range of fluctuation of visits or volume is the percentage of the average below which no truck should fall. For example, varVisits = 0.95 means that no truck may have a visit count of less than 95% of the average number of visits. Similarly applies to the volume and the loading capacity of the trucks.

For the following mathematical model we define the x_{ik} decision binary variable, which takes the value 1 when the client i is assigned to the truck k and 0 if not, $x_{ik} \in \{0,1\}$. Based on the above, we construct the following mixed-integer linear programming model:

$$\min \sum_{ik} c_{ik} x_{ik} \quad (1)$$

$$\sum_{k} x_{ik} = 1 \ \forall i \quad (2)$$

$$vT\ \bar{Q} \leq \sum_{i} x_{ik} q_i f_i \leq (2 - vT)\ \bar{Q}\ \forall k \quad (3)$$

$$vV\,\bar{V} \leq \sum_{i} x_{ik}\, f_i \leq (2 - vV)\,\bar{V}\ \forall k \qquad (4)$$

Function (1) corresponds to the objective function of the mathematical model and aims to minimize the assignment cost of customers to trucks. Constraint (2) guarantees that each customer will be assigned to one truck. The inequality (3) controls that the turnover to be disbursed by each truck, after the assignment, must not be outside the specified turnover limits, and the inequality (4) checks that the number of visits will also not violate the respective thresholds.

By resolving the above model, customers have been assigned to trucks in a way that respects two conditions that may not have been met in the previous step: a) the number of visits each trcuks will be required to make throughout the time be less than 95% of the average and at the same time b) the volume that will disburse from the orders each truck is not less than 95% of the average among the trucks. In addition, the algorithm maximizes the geographic distribution of customers to truck, minimizing the number of clients assigned to clusters different from those assigned to Step 3.

If the cost of the solution returned is zero, this means that there is no customer assigned to another cluster beyond what was initially assigned to Step 3. Otherwise, two cases may have occurred: a) geographic overlaps will be shown to be corrected in the next Step 5 of the algorithm and / or b) those customers assigned to another cluster were within the cluster boundaries and thus did not create any overlap.

Note that this mathematical model is solved by the branch and bound method using the Feasibility Pump[1] method, which is applied for 100 iterations. The current step 4 has been adapted to stop at the first integer solution. If more time to resolve this step is available, suggest the adaptation of the COIN-OR solving algorithm by setting the solution's optimization tolerance interval e.g. 100% or less in order to obtain an even better solution.

This step simulates the correction actions of the routing planner to the first routing plan generated in the previous step. The routing planner always studies the initial defined plan and investigates its potential feasibility to be implemented. If the trucks are over loaded or if the trucks have to visit a high number of customers, the routing planner tries to balance the volume and the number of visiting points between the trucks.

2.5 Step 5: Re-assignment of customers to a different salesman.

This step is triggered when overriding areas of responsibility are detected in the previous step. We believe that each customer puts a request for a truck to another truck due to being a customer of a truck in the area of responsibility of another truck. In order to find any overlapping, compare the assigned customers from Step 4 and those that came up with the reference axis and the process of defining the responsibility areas by scanning this axis in Step 3. Those who are in Step 4 in the responsibility area are different than those in Step 3, they create the transfer requests. A transfer request is the transfer of a customer placed in a different responsibility area in its original one. This process creates the following table of transitions:

[1] Fischetti, L. Bertacco, and A. Lodi. A feasibility pump heuristic for general mixed-integer problems. Technical report, Universita di Bologna – D.E.I.S. – Operations Research, 2005

Customer	Salesman		Mode
	From	To	
5	0	2	False
18	0	3	False
187	0	1	False
…………	…………	…………	…………

Where each customer requests a transfer from one truck to another and the last column shows the situation in which this request was found after the mathematical model below has been resolved.

For the following mathematical model, we insert the binary variable z_s where s is the request (the line of the above table) with 1 if the request is finally accepted and 0 if not, $z_s \in \{0,1\}$.

The following table presents the definitions of indexes and parameters:

Δείκτης	**Περιγραφή**
s	Index of re-assignment including customers i and truck from/to
k	Index of truck
Parameters	**Description**
Q_{min}/Q_{max}	Max and min volume
V_{min}/V_{max}	Max and min number of visits
Δ^+	Group of requests for re-assigment of customers assigmened to truck (Column «to salesman»)
Δ^-	Group of requests for re-assigment of customers assigmened to truck (Column «from salesman»)
Q_k	Total volume resulted in step 4
V_k	Total number of visits resulted in step 4
q_s	Amount corresponding to the customer requesting the transfer from one truck to another and corresponding to the request s.
f_s	Frequency of visit corresponding to the customer requesting transfer from one truck to another and corresponding to the request s.

To solve this problem we construct the following mixed-integer linear programming model:

$$\max \sum_{i} z_s \tag{5}$$

$$Q_{min} - Q_k \leq \sum_{s \in \Delta^+} z_s q_s f_s - \sum_{s \in \Delta^-} z_s q_s f_s \leq Q_{max} - Q_k \;\forall k \tag{6}$$

$$V_{min} - V_k \leq \sum_{s \in \Delta^+} z_s f_s - \sum_{i \in \Delta^-} z_s f_s \leq V_{max} - V_k \;\forall k \tag{7}$$

Function (5) is the objective function of the mathematical model which seeks to maximize the satisfaction of the requested re-assigmnets. Inequalities (6) and (7) guarantee that the quantities of orders and the number of post-shipment visits must meet the limits we have set for each truck (i.e. not below 95% of the average and not more than 105% of this).

To solve this model we also apply branch and bound methods with Feasibility Pump and set the solution's optimization tolerance range to 10%. The optimization tolerance interval of the solution is set at this rate because we keep a good balance in the quality of the solution that will occur and in the time it takes to converge.

At this point the algorithm checks whether all the number of requests is satisfied or not. If not, we have two options, depending on the user's preference:

- If the user has stated that he / she wants an absolute geographical distribution, then we meet the demands of unsatisfied customers at the expense of the balance of volume and the total number of visits,
- If the user does not want an absolute geographical distribution, then we leave the customers as they are by observing the constraints of volume and number of visits by conceding to the geographical distribution.

One such example is shown in the following figure where the two red customers (the red squares) are the one in the green responsibility area and the other the red truck, creating a 2-point geographic distribution problem.

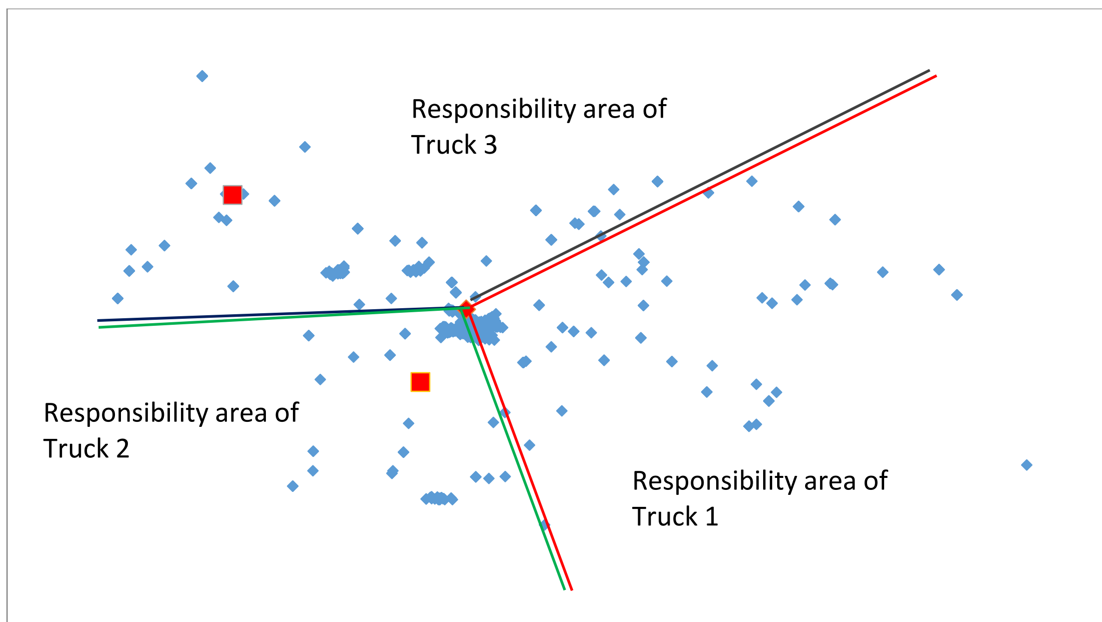

Figure 5: Example of problematic regions of responsibility

Another example where the user has not selected a geographical breakdown as absolute criterion is shown below. In the figure there are some overlaps resulting from the user choosing to fully meet the volume criteria and the number of visits.

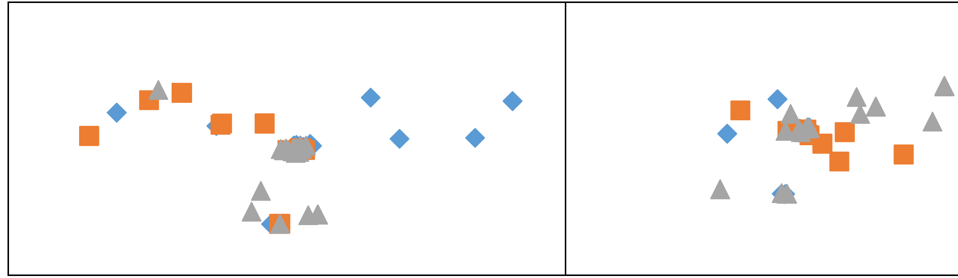

Figure 6: Example of absolute satisfaction of volume and number of visits where the geographical coverage is not totally satisfied

At the end of this step, the program creates a <NAME> -Assignment.txt file listing each client and the seller to whom it was assigned. The file shows the results as follows:

Customer	Vehicle
1	3
2	3
5	3
6	2
7	2
8	2
9	3
10	2
…………………………….	

The previous step simulates the correction actions of the routing planner to the first routing plan generated in the 4th step. The current step revise this plan trying at the same time to minimize the total distance travelled by the trucks. At the end of this step the routing planner has trucks and orders assigned to trucks.

2.6 Step 6: Optimal routing plan of trucks

In step is the final step of the simulation tool. Orders assigned to trucks should be planned to be delivered in the optimal order/sequence. For this reason the classical nearest insertion algorithm is coded and implemented to define the optimal routes of the trucks.

The algorithm includes the following steps

Step 1. Start with a sub-graph consisting of node i only.

Step 2. Find node r such that c_{ir} is minimal and form sub-tour i-r-i.

Step 3. (Selection step) Given a sub-tour, find node r not in the sub-tour closest to any node j in the sub-tour; i.e. with minimal c_{rj}

Step 4. (Insertion step) Find the arc (i, j) in the sub-tour which minimizes $c_{ir} + c_{rj} - c_{ij}$. Insert r between i and j.

Step 5. If all the nodes are added to the tour, stop. Else go to step 3

3 Conclusion

The algorithm developed consists of 6 basic steps: Step 1: Calculate the center of weight (points), Step 2: Statistical analysis, Step 3: Scanning, Step 4: Problem re-assignment of customers to salesmen, Step 5: Re-assignment of customers to a different salesman, and Step 6: optimal routing plan of trucks. This simulation tool has been applied to the examples defined during the real life practice of GYR service. The obtained routing plans using this simulation tool have been compared with the obtained routing plans by the GYR service which actually were the plans implemented by the demonstrators.

4 Annex

In this annex the code including the assignment of orders to trucks is presented:

```
#include <iostream>

#include <fstream>

#include <windows.h>

#include <stdio.h>

#include <time.h>

#include <string>

#include <sstream>

#include <vector>

#include <numeric>

#include <algorithm>

#include "OsiClpSolverInterface.hpp"

#include "CoinPackedVector.hpp"

#include "CoinPackedMatrix.hpp"

#include "CbcModel.hpp"

#include "CbcHeuristic.hpp"

#include "CbcHeuristicFPump.hpp"

#include "CbcCompareActual.hpp"

#include "CbcBranchDynamic.hpp"

#include "CbcHeuristicRINS.hpp"

#include "CbcHeuristicPivotAndFix.hpp"


using namespace std;


// I am using this bigM once, in the Daily scheduling problem

const double bigM = 1000;


// The calc_Index function is called to transform a 2D matrix to 1D array

// For instance, instead of calling the element A[5][7] of a 10x10 matrix
```

```
// we use calc_Index(5, 7, 10) which returns 57 and have the element B[57]
// We use this function because we want to avoid matrices of more than
// one dimension
int calc_Index(int i, int j, int J)
{
	return (i*J+j);
}

// The same function for a 3D matrix. It turns the 3D matrix into a 1D array
int calc_Index(int i, int j, int k ,int J, int K)
{
	return (i*J*K+j*K+k);
}

// This structure is the structure of customer
// It includes everything read from the file and some additional info,
// such as to which vehicle this customer is assigned to, to which cluster
// and also what's his phi angle for the clustering of the first round
struct Customer
{
	int id, iter; // id as read from file. Iter iterates from 0 to nCustomers, so that Customer[i] is the i-th node read
	double x, y; // cordinates
	double q; // turnover
	int f; // frequency
	int assignedTo; // to which vehicle this node is assigned to
	int clusteredTo; // to which cluster is this node assigned to
	double phi;
};
```

```
// The only use of this class is to be able to search a vector of structs
// and return true if the ID of the item is found in the vector of structs
class MatchesID
{
	int _ID;

public:
	MatchesID(const int &id) : _ID(id) {}
	template<class ItemType>
	bool operator()(const ItemType &item) const
	{
		return item.id == _ID;
	}
};



// This structure is a a simple structure of two numbers
// For instance it is used to store the (long, lat) of a customer
// or the two borders of a cluster region in angles, for instance
// (from a=30 degress, to b=50 degrees)
// It is also used to say which two clusters should be twinned.
// Clusters are twinned when their number of assigned customers
// is small
template <typename T>
struct TwoNumbers
{
	T a;
	T b;
};
```

```
// This function is used in conjuction with a vector to return
// a vector sorted by angle. First will be placed the smaller phi
bool smallerPhi(const Customer &a, const Customer &b)
{
	return (a.phi < b.phi);
}

// This function is used to sort customers starting from the one with highest turnover
// It is used by the function performAnalysis when reporting bizarre data
bool higherTurnover(const Customer &a, const Customer &b)
{
	return (a.q > b.q);
}
// This function is used in conjuction with a vector to return
// a vector sorted by "iter". Everytime we call this function
// we ensure that the vector Customers (for instance) is sorted
// as it was read from the file
bool asReadInFile(const Customer &a, const Customer &b)
{
	return (a.iter < b.iter);
}

// We use this struct for the requests of wrongly assigned customers to vehicle
// to change clusters. For instance customer=10, requests to leave from
// vehicle from=2 and go to vehicle to=3. His request has been satisfied
// if fixed=true. In this case, the fixed is a boolean that shows whether the
// request is fulfilled. This struct is used at the fixOverlaps step of the algorithm
// We may use the same struct with fixed being double, in another context. This is
// why it is templated
```

```
template <typename T>
struct FromTo
{
        int customer, from,  to;
        T fixed;
};

// This struct is used for the centroids.
// Every centroid has an id, a x and a y coordinate
// Q and V that has been assigned to it and a phi_start
// and a phi_end angle to mark its region
struct Centre
{
        int id;
        double x, y; // cordinates
        double Q, V; // max and min turnover/visits accepted
        double phi_start, phi_end; // ranges from phi_start to phi_end
};

// This is the struct of the vehicle. A vehicle has an ID, a quantity Q
// (which is the turnover generated), a number of visits V, a min/max
// turnover (Qmin/Qmax) and a min/max number of visits that it should
// achieve
struct Vehicle
{
        int id;
        double Q, V;
        double Qmin, Qmax, Vmin, Vmax;
};
```

```
// This function performs a statistical analysis prior to solving the problem

void performAnalysis(vector<Customer>&, Centre, ofstream&);

// This function performs the radar circular scanning

void radarScan(vector<Customer>&, vector<Vehicle>&, Centre&, ofstream&);

// This function assigns customers to vehicles

int assignCustomersToVehicles(vector<Customer>&, vector<Vehicle>&, double, double,
vector<FromTo<bool>>&, ofstream&);

// This function fixes overlaps between customers assigned to wrong vehicles

void fixOverlaps(vector<Customer>&, vector<Vehicle>&, vector<FromTo<bool>>&, bool,
ofstream&);

// This function is called to perform the k-means clustering

// It requires the customers, the customers assigned to vehicles, the information on which
clusters

// should be twinned, the current vehicle to cluster k, the number of Vehicles, Days and
Clusters;

void kmeans(vector<Customer>&, vector<Customer>&, vector <TwoNumbers<int> >&, int,
int, int, int);

// This function constructs and runs a MILP which schedules the daily assignments of
customers to vehicles/

// It requires the struct of customers, the structs if vehicles, the number of clusters, the
number of days and a stream for the Ouput file

int assignCustomersToDays(vector<Customer>&, vector<Vehicle>&, int, int, string, string,
string, ofstream&);


int main(int argc, char *argv[])

{

    system("CLS");

    srand(unsigned(time(NULL)));

    clock_t tStart = clock();

    // Data

    int nCustomers, nVehicles, nDays;

    bool geoDistribAbsolutePriority = false;
```

```
string NAME; // This is the name of the problem run following which the log file will be created

string sCust, sDP, sCP;

// Read file with Preferences

const std::string fnP = "Preferences.txt";

//  Input streams

std::ifstream fP;      // file stream for customers

// Open files

fP.open(fnP.c_str()); // open file and create stream for customers

// Check if files exist

if(!fP)

{

        cout << "=====================================================" << endl

                << "| For AREA MANAGER to run properly, a file named " << endl

                << "| Preferences.txt is required. This file will look like:" << endl

                << "| " << endl

                << "| Name: MyInstance" << endl

                << "| nDays: 20" << endl

                << "| nVehicles: 12" << endl

                << "| varTurnover: 0.95" << endl

                << "| varVisits: 0.95 " << endl

                << "| varDayVisits: 0.95 " << endl

                << "| Is_the_geographical_distribution_your_absolute_priority_(YES/NO): NO" << endl

                << "| InputFile_Days_Patterns: Input-DP.txt" << endl

                << "| InputFile_Customers_Patterns: Input-CP.txt" << endl

                << "| " << endl

                << "| Please create that file and place it at the" << endl

                << "| same directory with AREA MANAGER" << endl

                << "| Please enter your preferences right after the colons" << endl
```

```
                    << "| making sure that text before the colons is" << endl
                    << "| identical to the text seen above." << endl
                    << "| If not, then AREA MANAGER will not run properly" << endl
                    << endl
                    << "| Press any key to exit;" << endl
                    <<
"========================================================" << endl;
                    getchar();
            return -1;
        }
        double varVisits, varTurnover;
        string line;
        while(std::getline(fP, line))
        {
            std::istringstream iss(line);
            size_t foundVarVisits = line.find("varVisits:");
            if (foundVarVisits!=string::npos)
            {
                    string tmp;
                    stringstream ss(line);
                    iss >> tmp >> varVisits;
                    continue;
            }
            size_t foundVarTurnover = line.find("varTurnover:");
            if (foundVarTurnover!=string::npos)
            {
                    string tmp;
                    stringstream ss(line);
                    iss >> tmp >> varTurnover;
                    continue;
```

```
}
size_t foundnDays = line.find("nDays:");
if (foundnDays!=string::npos)
{
        string tmp;
        stringstream ss(line);
        iss >> tmp >> nDays;
        continue;
}
size_t foundnVehicles = line.find("nVehicles:");
if (foundnVehicles!=string::npos)
{
        string tmp;
        stringstream ss(line);
        iss >> tmp >> nVehicles;
        continue;
}
size_t foundNAME = line.find("Name:");
if (foundNAME!=string::npos)
{
        string tmp;
        stringstream ss(line);
        iss >> tmp >> NAME;
        continue;
}
size_t foundInFileCust = line.find("InputFile_Customers:");
if (foundInFileCust!=string::npos)
{
        string tmp;
        stringstream ss(line);
```

```
                iss >> tmp >> sCust;
                continue;
            }
            size_t foundInFileDP = line.find("InputFile_Days_Patterns:");
            if (foundInFileDP!=string::npos)
            {
                string tmp;
                stringstream ss(line);
                iss >> tmp >> sDP;
                continue;
            }
            size_t foundInFileCP = line.find("InputFile_Customers_Patterns:");
            if (foundInFileCP!=string::npos)
            {
                string tmp;
                stringstream ss(line);
                iss >> tmp >> sCP;
                continue;
            }
            size_t foundGeoDistribution =
line.find("Is_the_geographical_distribution_your_absolute_priority_(YES/NO):");
            if (foundGeoDistribution!=string::npos)
            {
                string tmp;
                stringstream ss(line);
                iss >> tmp >> tmp;
                geoDistribAbsolutePriority = (tmp == "YES" || tmp == "1" || tmp ==
"TRUE");
                continue;
            }
        }
```

```
string NAMELOG = NAME + "-Progress.log";
const char *fnNameLog = NAMELOG.c_str();
ofstream fLog(fnNameLog);// std::ios_base::app);
string NAMERESULTS = NAME + "-Results.log";
string NAMECUSTASSIGN = NAME + "-Assignment.txt";
const char *fnNameAssign = NAMECUSTASSIGN.c_str();

cout << "=======================================================" << endl
    << "| AREA MANAGER is running... " << endl
    << "| using the input preferences found in Preferences.txt:" << endl
    << "|" << endl
    << "| Name: " << NAME << endl
    << "| nDays: " << nDays << endl
    << "| nVehicles: " << nVehicles << endl
    << "| varTurnover: " << varTurnover << endl
    << "| varVisits: " << varVisits << endl
    << "| Is_the_geographical_distribution_your_absolute_priority_(YES/NO): ";
if(geoDistribAbsolutePriority)
    cout << "YES" << endl;
else
    cout << "NO" << endl;
cout << "| InputFile_Customers: " << sCust << endl
    << "| InputFile_Days_Patterns: " << sDP << endl
    << "| InputFile_Customers_Patterns: " << sCP << endl
    << "|" << endl
    << "| If you do not agree with some of the data or" << endl
    << "| preferences above, please stop execution (ctrl+c)," << endl
    << "| modify the file Preferences.txt and retry " << endl
    << "| " << endl
```

```
        << "| Printing log information and results at files named:" << endl
        << "| " << NAMELOG << " and " << NAMERESULTS << endl
        << "| Please check progress in these files" << endl
        << "|" << endl
        << "| When finished, this window will close" << endl
        <<
"===========================================================" << endl;

    int problemSolved;
    bool firstPass = true;
    // Demand per customer
    // Number of total visits throughout horizon
    int nVisits = 0;
    // Coordinates for each customer - serve to calculate distances
    vector<TwoNumbers<double>> longlat;
    // Customers (this struct includes coorinates -x,y,phi, demand, vehicle and cluster
    vector<Customer> customer;
    // Vehicles
    vector<Vehicle> vehicle;
    // Input files
    const std::string fnC = sCust;//"Input-Customers.txt";
    //  Input streams
    std::ifstream fC;      // file stream for customers
    // Open files
    fC.open(fnC.c_str()); // open file and create stream for customers
    // Check if files existf
    if(!fC)
    {
        system("CLS");
        cout                                                        <<
"===========================================================" << endl
```

```
                    << "| AREA MANAGER stopped, because " << endl
                    << "| " << fnC << " file has not been found" << endl
                    << "| " << endl
                    << "| Please make the encessary changes to the" << endl
                    << "| Preferences.txt file and retry" << endl
                    << "|" << endl
                    << "| Press any key to exit;" << endl
                    <<
"=====================================================" << endl;
            getchar();
            return 0;
        }
        ifstream check_sDP(sDP.c_str());
        if(!check_sDP)
        {
            system("CLS");
            cout                                                          <<
"=====================================================" << endl
                    << "| AREA MANAGER stopped, because " << endl
                    << "| " << sDP << " file has not been found" << endl
                    << "| " << endl
                    << "| Please make the necessary changes to the" << endl
                    << "| Preferences.txt file and retry" << endl
                    << "|" << endl
                    << "| Press any key to exit;" << endl
                    <<
"=====================================================" << endl;
            getchar();
            return 0;
        }
```

```
// Buffer variable
std::string tmp;
// Variable Q will store total turnover
double Q = 0;
// Read file
// Here I read the first line of headers
getline(fC, line);
string Pelatis,  X, Y, ArxikiAksia, Sixnotita;
int iter = -1;
Centre barycentre = {0, 0, 0, 0, 0};
while(std::getline(fC, line))
{
        std::istringstream iss(line);
        if(!line.empty())
        {
                iss >> Pelatis >> X >> Y >> ArxikiAksia >> Sixnotita;
                TwoNumbers<double> TN = {atof(X.c_str()), atof(Y.c_str())};
                longlat.push_back(TN);
                Customer next_customer = {atoi(Pelatis.c_str()), ++iter, atof(X.c_str()),
atof(Y.c_str()), atof(ArxikiAksia.c_str()), atoi(Sixnotita.c_str()), -1, -1, 0};
                customer.push_back(next_customer);
                barycentre.x += longlat[iter].a;
                barycentre.y += longlat[iter].b;
                nVisits +=  atoi(Sixnotita.c_str());
                Q += atoi(Sixnotita.c_str())*atof(ArxikiAksia.c_str());
        }
}
nCustomers = customer.size();
for(int k=0; k<nVehicles; k++)
{
```

```
        Vehicle aVehicle = {k, 0, 0, varTurnover*Q/nVehicles, (2-varTurnover)*Q/nVehicles, varVisits*nVisits/nVehicles, (2-varVisits)*nVisits/nVehicles};
        vehicle.push_back(aVehicle);
    }
    barycentre.x /= nCustomers;
    barycentre.y /= nCustomers;

    performAnalysis(customer, barycentre, fLog);

    fLog << endl <<  "Instance being solved" << endl;
    fLog << "# of customers................ " << nCustomers << endl;
    fLog << "# of vehicles................. " << nVehicles << endl;
    fLog << "Var on visits/vehicle......... " << varVisits*100 << "%"  << endl;
    fLog << "Var on turnover............... " << varTurnover*100 << "%"  << endl;
    fLog << endl;
    do
    {
        clock_t tCluster = clock();
        clock_t tRadar = clock();
        radarScan(customer, vehicle, barycentre, fLog);
        fLog << "Radar Scan lasted " << (double)(clock() - tRadar)/CLOCKS_PER_SEC << "sec" << endl;
        // Undo sorting by angle.
        // Sort customers based on the rank they were read initially from file (iter)
        // The following will NOT  work, unless  customers are sorted as originally
        // ranked when the file was read.
        std::sort(customer.begin(), customer.end(), asReadInFile);
        vector<FromTo<bool>> ft;
        clock_t tAssignCusttoVeh = clock();
        int nOverlaps = assignCustomersToVehicles(customer, vehicle, varVisits, varTurnover, ft, fLog);
```

```
                fLog << "Assignment of customers to vehicles lasted " << (double)(clock() -
tAssignCusttoVeh)/CLOCKS_PER_SEC << "sec" << endl;

                fLog << "During assignment of customers to vehicles " << nOverlaps << "
overlaps were identified" << endl;

                // If there exist any overlaps, identify them and fix them

                if(nOverlaps)

                {

                        clock_t tFixing = clock();

                        // IMPORTANT: For the fixOverlaps function to work, it is necessary
to

                        // have the customers sorted by angle.

                        std::sort(customer.begin(), customer.end(), smallerPhi);

                        fixOverlaps(customer, vehicle, ft, geoDistribAbsolutePriority, fLog);

                        fLog << "Fixing overlaps lasted " << (double)(clock() -
tFixing)/CLOCKS_PER_SEC << "sec" << endl;

                        // Undo sorting by angle.

                        // Sort customers based on the rank they were read initially when file
was read (by iter)

                        std::sort(customer.begin(), customer.end(), asReadInFile);

                }

                ofstream fAssign(fnNameAssign);

                fAssign << "Customer" << " Vehicle" << endl;

                for(int i=0; i<nCustomers; i++)

                        fAssign << customer[i].id << "\t" << customer[i].assignedTo+1 <<
endl;

                fLog << endl << "---> File " << NAMECUSTASSIGN << " has been created. It
contains the assignments of customers to vehicles" << endl;

                int nClusters = 5;

                clock_t tAsCustToDays = clock();

                problemSolved = assignCustomersToDays(customer, vehicle, nClusters,
nDays, NAME, sDP, sCP, fLog);

                fLog << "Assignment of customers to days lasted " << (double)(clock() -
tAsCustToDays)/CLOCKS_PER_SEC << "sec" << endl;
```

```
		if(problemSolved == 1)
		{
			fLog << endl << endl;
			fLog << "Number of visits and value of order collected per vehicle throughout the time horizon" << endl;
			for(int k=0; k<nVehicles; k++)
				fLog << "Vehicle " << k+1 << "\t Visits: " << vehicle[k].V << "\t Value of order: " << vehicle[k].Q << endl;
			fLog << endl;
			fLog << endl << "Total runtime " << (double)(clock() - tStart)/CLOCKS_PER_SEC << "sec" << endl;
		}
	} while(problemSolved == -1);
}


void fixOverlaps(vector<Customer> &customer, vector<Vehicle> &vehicle, vector<FromTo<bool>> &ft, bool geoDistribAbsolutePriority, ofstream &fLog)
{
	fLog << endl << "Fixing overlaps " << endl;
	bool successfulFix = true;
	int nCustomers = customer.size();
	int nVehicles = vehicle.size();
	int nOverlaps = ft.size();
	double *q = new double [nCustomers];
	int *f = new int [nCustomers];
	for(int i=0; i<nCustomers; i++)
	{
		int ii = customer[i].iter;
		double qq = customer[i].q;
		int ff = customer[i].f;
```

```
        q[ii] = qq;
        f[ii] = ff;
}
int noVar = -1;
int *varZ;
double *objective, *col_lb, *col_ub, *row_lb, *row_ub;
int nCols = nOverlaps;
int nRows = nVehicles+nVehicles;
varZ      = new int    [nCols];
objective  = new double [nCols];
col_lb     = new double [nCols];
col_ub     = new double [nCols];
row_lb     = new double [nRows];
row_ub     = new double [nRows];
OsiSolverInterface *model = new OsiClpSolverInterface();
for(int i=0; i<nOverlaps; i++)
{
        varZ[i] = ++noVar;
        col_lb[noVar] = 0;
        col_ub[noVar] = 1;
        objective[noVar] = -1;
}
CoinPackedMatrix * matrix = new CoinPackedMatrix(false, 0,0);
matrix->setDimensions(0, noVar+1);
int noCtr = -1;
for(int k=0; k<nVehicles; k++)
{
        ++noCtr;
        row_lb[noCtr] = vehicle[k].Qmin-vehicle[k].Q;
        row_ub[noCtr] = vehicle[k].Qmax-vehicle[k].Q;
```

```
                CoinPackedVector row;
                for(int i=0; i<nOverlaps; i++)
                {
                        int customerWhoRequests = ft[i].customer;
                        if(ft[i].from == k)
                                row.insert(varZ[i], -1*q[customerWhoRequests]*f[customerWhoRequests]);
                        else if(ft[i].to == k)
                                row.insert(varZ[i], q[customerWhoRequests]*f[customerWhoRequests]);
                }
                matrix->appendRow(row);
        }
        for(int k=0; k<nVehicles; k++)
        {
                ++noCtr;
                row_lb[noCtr] = vehicle[k].Vmin-vehicle[k].V;
                row_ub[noCtr] = vehicle[k].Vmax-vehicle[k].V;
                CoinPackedVector row;
                for(int i=0; i<nOverlaps; i++)
                {
                        int customerWhoRequests = ft[i].customer;
                        if(ft[i].from == k)
                                row.insert(varZ[i], -1*f[customerWhoRequests]);
                        else if(ft[i].to == k)
                                row.insert(varZ[i], f[customerWhoRequests]);
                }
                matrix->appendRow(row);
        }
        model->loadProblem(*matrix, col_lb, col_ub, objective, row_lb, row_ub);
        fLog << "Number of variables: " << model->getNumCols() << endl
```

```
        <<  "Number of constraints: " << model->getNumRows() << endl
        << "Number of variables: " << model->getNumElements() << endl;
model->setHintParam(OsiDoReducePrint,true, OsiHintDo);
for(int i=0; i<nOverlaps; i++)
        model->setInteger(varZ[i]);
CbcModel BB(*model);
CbcHeuristicFPump *heur1 = new CbcHeuristicFPump();
BB.addHeuristic(heur1);
BB.setLogLevel(0);
BB.setAllowableFractionGap(0.1);
BB.branchAndBound();
delete heur1;
if(BB.isProvenOptimal())
{
        int nOverlapsFixed = 0;
        const double *solution = BB.bestSolution();
        for(int i=0; i<nOverlaps; i++)
        {
                if(solution[varZ[i]] > 10e-5)
                {
                        ft[i].fixed = true;
                        nOverlapsFixed++;
                        int from = ft[i].from;
                        int to = ft[i].to;
                        for(int ii=0; ii<nCustomers; ii++)
                        {
                                if(customer[ii].iter == ft[i].customer)
                                {
                                        customer[ii].assignedTo = ft[i].to;
                                        vehicle[from].V -= customer[ii].f;
```

```
                        vehicle[from].Q -= customer[ii].f*customer[ii].q;
                        vehicle[to].V += customer[ii].f;
                        vehicle[to].Q += customer[ii].f*customer[ii].q;
                    }
                }
            }
            else
            {
                if(!geoDistribAbsolutePriority)
                    ft[i].fixed = false;
                else
                {
                    // If geographic distribution is of absolute priority, then
change vehicles for the wrongly assigned
                    //fLog << endl << "Since you have asked for absolute
                    ft[i].fixed = true;
                    int from = ft[i].from;
                    int to = ft[i].to;
                    for(int ii=0; ii<nCustomers; ii++)
                    {
                        if(customer[ii].iter == ft[i].customer)
                        {
                            customer[ii].assignedTo = ft[i].to;
                            vehicle[from].V -= customer[ii].f;
                            vehicle[from].Q                    -=
customer[ii].f*customer[ii].q;
                            vehicle[to].V += customer[ii].f;
                            vehicle[to].Q                      +=
customer[ii].f*customer[ii].q;
                        }
                    }
```

```
                        }
                    }
                }
            }
            fLog << nOverlaps << " overlaps were fixed" << endl;
            delete [] q;
            delete [] f;
            delete model;
            delete matrix;
            delete [] varZ;
            delete [] objective;
            delete [] col_lb;
            delete [] col_ub;
            delete [] row_lb;
            delete [] row_ub;
}

        int assignCustomersToDays(vector<Customer> &customer, vector<Vehicle> &vehicle, int nClusters, int nDays, string NAME, string sDP, string sCP, ofstream &fLog)
        {
                fLog << endl << "Assignment of customers to days" << endl;
                std::sort(customer.begin(), customer.end(), asReadInFile);
                int successfulAssignment = 1;
                int nOverlaps = 0;
                string NAMERESULTS = NAME + "-Results.txt";
                const char *fnResults = NAMERESULTS.c_str();
                ofstream fout(fnResults);
                int nVehicles = vehicle.size();
                int nVisits = (int) vehicle[0].Vmax;
                vector<int> maxOvertime;
```

```
vector<int> M;
vector<int> freqOfPattern;
int nFailures;
std::ifstream fM;
std::ifstream fO;
fM.open(sDP.c_str());
fO.open(sCP.c_str());
// Continue reading the input files
int _r = -1;
string line;
while(std::getline(fM, line))
{
    std::istringstream iss(line);
    if(!line.empty())
    {
        string _freq;
        iss >> _freq;
        char *dest = new char [10];
        if(std::strcmp(_freq.c_str(),dest) != 0)
        {
            freqOfPattern.push_back(atoi(_freq.substr(2,2).c_str()));
            _r++;
            for(int t=0; t<nDays; t++)
            {
                int isDayCompatibleWithPattern;
                iss >> isDayCompatibleWithPattern;
                if(isDayCompatibleWithPattern != 1 &&
isDayCompatibleWithPattern != 0)
                    fLog << "File not read properly" << endl;
                int index = calc_Index(_r, t, nDays);
```

```
                        M.push_back(isDayCompatibleWithPattern);
                    }
                }
                delete [] dest;
            }
        }
        int nPatterns = M.size()/nDays;
        for(int k=0; k<nVehicles; k++)
        {
            clock_t tVeh = clock();
            fLog << "For vehicle " << k+1 << ":" << endl;
            if(successfulAssignment == 1)
                nFailures = 0;
            vector<double> q;
            vector<int> f;
            // For every k we will solve an LP
            // But the variables relating to the customers
            // should correspond only to the customers assigned
            // to the vehicle k. We thus need a way to map
            // customers from the struct "customers" to the variables
            // 1...nCustomers*nPatterns, where nCustomers  are the
            // customers assigned to vehicle k
            vector<int> translate;
            // We assume that customers are sorted by ID.
            // translate[i] will contain the id of the customer
            // behind the varX[i] variable
            vector<Customer> CustomersAssignedToVehicle;
            for(unsigned int i=0; i<customer.size(); i++)
            {
            if(customer[i].assignedTo == k)
```

```
                {
                CustomersAssignedToVehicle.push_back(customer[i]);
                translate.push_back(customer[i].id);
                f.push_back(customer[i].f);
                q.push_back(customer[i].q);
                }
        }
    int nCustomers = q.size();
    // Cluster then assign.
    // 1st step: Clustet, using the k-means
    vector <TwoNumbers<int> > twinClusters;
    kmeans(customer, CustomersAssignedToVehicle, twinClusters, k, nVehicles,
nDays, nClusters);
    // Vector L will contain values "true" if i-customer
    // belongs in l-cluster of k-vehicle and "false" if not.
    vector<bool> L;
    for(int i=0; i<nCustomers*nClusters; i++)
        L.push_back(false);
    for(int i=0; i<nCustomers; i++)
        {
        int j = CustomersAssignedToVehicle[i].clusteredTo;
        int index = calc_Index(i, j, nClusters);
        L[index] = true;
        int ii = CustomersAssignedToVehicle[i].iter;
        }
    // 2nd Step: Assign using MILP
    // Assistant matrix O: Shows whether pattern r is compatible with customer i
    vector<int> O;
    for(int r=0; r<nPatterns; r++)
    {
```

```
                int ii = -1;
                for(unsigned i=0; i<customer.size(); i++)
                {
                        if(customer[i].assignedTo == k)
                        {
                                if(f[++ii] == freqOfPattern[r])
                                        O.push_back(1);
                                else
                                        O.push_back(0);
                        }
                }
        }
        int noVar = -1;
        int *varX, *varZ, *varUunder, *varUover, *varV, *ctrDayVisits;
        double *objective, *col_lb, *col_ub, *row_lb, *row_ub;
        int nCols = nCustomers*nPatterns+nDays*nClusters+nDays+nDays*twinClusters.size();
        int nRows = nCustomers+nDays+2*nDays*nClusters+(1+twinClusters.size())*nDays;
        varX      = new int   [nCustomers*nPatterns];
        varZ      = new int   [nDays*nClusters];
        varUunder = new int   [nDays];
        varUover  = new int   [nDays];
        varV      = new int   [nDays*twinClusters.size()];
        ctrDayVisits = new int [nDays];
        objective  = new double [nCols];
        col_lb     = new double [nCols];
        col_ub     = new double [nCols];
        row_lb     = new double [nRows];
        row_ub     = new double [nRows];
        // I want to be sure that all the above are zero'ed just in case
```

```
for(int col=0; col<nCols; col++)
    {
    objective[col] = 0;
    col_lb[col] = 0;
    col_ub[col] = 0;
    }
for(int row=0; row<nRows; row++)
    {
    row_lb[row] = 0;
    row_ub[row] = 0;
    }

OsiSolverInterface *model = new OsiClpSolverInterface();
for(int i=0; i<nCustomers; i++)
    {
    for(int r=0; r<nPatterns; r++)
        {
        int index = calc_Index(r, i, nCustomers);
        if(O[index])
            {
            int index = calc_Index(i, r, nPatterns);
            varX[index] = ++noVar;
            col_lb[noVar] = 0;
            col_ub[noVar] = 1;
            objective[noVar] = 0;
            }
        }
    }
for(int t=0; t<nDays; t++)
    {
```

```
                for(int j=0; j<nClusters; j++)
                    {
                    int index = calc_Index(t, j, nClusters);
                    varZ[index] = ++noVar;
                    col_lb[noVar] = 0;
                    col_ub[noVar] = 1;
                    objective[noVar] = 1;
                    }
                }
        for(int t=0; t<nDays; t++)
                {
                varUunder[t] = ++noVar;
                col_lb[noVar] = 0;
                col_ub[noVar] = (0.3+(double) nFailures/20.0)*vehicle[0].Vmin/nDays;
                objective[noVar] = 1;
                varUover[t] = ++noVar;
                col_lb[noVar] = 0;
                col_ub[noVar] = (0.3+(double) nFailures/20.0)*vehicle[0].Vmin/nDays;
                objective[noVar] = 1;
                }
        for(int t=0; t<nDays; t++)
                {
                for(unsigned j=0; j<twinClusters.size(); j++)
                    {
                    int index = calc_Index(t, j, twinClusters.size());
                    varV[index] = ++noVar;
                    col_lb[noVar] = 0;
                    col_ub[noVar] = 1;
                    objective[noVar] = 0;
```

```
                }
        }
CoinPackedMatrix * matrix = new CoinPackedMatrix(false, 0,0);
matrix->setDimensions(0, noVar+1);
int noCtr = -1;
// Constraint #1: Assign customers to patterns
for(int i=0; i<nCustomers; i++)
        {
        ++noCtr;
        row_lb[noCtr] = 1;
        row_ub[noCtr] = 1;
        CoinPackedVector row;
        for(int r=0; r<nPatterns; r++)
                {
                int index = calc_Index(r, i, nCustomers);
                if(O[index])
                        {
                        int index = calc_Index(i, r, nPatterns);
                        row.insert(varX[index], 1);
                        }
                }
        matrix->appendRow(row);
        }
// Constraint #2: Number of visits should be bounded but
// to ensure feasibility we allow a slack variable U(t) per day
for(int t=0; t<nDays; t++)
        {
        ctrDayVisits[t] = ++noCtr;
        row_lb[noCtr] = vehicle[0].Vmin/nDays;
        row_ub[noCtr] = vehicle[0].Vmax/nDays;
```

```
        CoinPackedVector row;
        for(int i=0; i<nCustomers; i++)
            {
            for(int r=0; r<nPatterns; r++)
                {
                int index1 = calc_Index(r, i, nCustomers);
                int index2 = calc_Index(r, t, nDays);
                if(O[index1] && M[index2])
                    {
                    int index = calc_Index(i, r, nPatterns);
                    row.insert(varX[index], 1);
                    }
                }
            }
        row.insert(varUover[t], -1);
        row.insert(varUunder[t], 1);
        matrix->appendRow(row);
        }
    // Constraint #3: Assign clusters to days
    for(int t=0; t<nDays; t++)
        {
        // First constraint:
        // Instead of z1+z2+z3+..+zn <= 1
        // substitute pairs of z which are twins with v
        // for instance if 1 and 2 are twins
        // and also say 4 and 6 are twins
        // then v1+z3+v2+...+zn <= 1
        ++noCtr;
        row_lb[noCtr] = 1;//-1*model->getInfinity();
        row_ub[noCtr] = 1;
```

```
CoinPackedVector row;
// If a cluster is not in twinsCluster
// then it is represented by the z variable.
// If it is then the two clusters of the pair are
// represented by the v variable.
for(int j=0; j<nClusters; j++)
	{
	bool isTwinned = false;
	for(unsigned jj=0; jj<twinClusters.size(); jj++)
		if(twinClusters[jj].a == j || twinClusters[jj].b == j)
			{
			isTwinned = true;
			break;
			}
		if(!isTwinned)
			{
			int index = calc_Index(t, j, nClusters);
			row.insert(varZ[index], 1);
			}
	}
for(unsigned j=0; j<twinClusters.size(); j++)
	{
	int index = calc_Index(t, j, twinClusters.size());
	row.insert(varV[index], 1);
	}
matrix->appendRow(row);
// Constraint #4: Link z to v
// Second constraint:
// For every pair of twins regulate whether they are allowed to take the value of 0 or 1
```

```
// For instance: z0+z1 <= 2u1
// OR if z0 appears in say two pairs then:
// z0 + z1 <= 2v1 + v2
// z1 + z2 <= v1 + 2v2
for(unsigned j=0; j<twinClusters.size(); j++)
        {
        // We run the list of twinClusters
        int p1 = twinClusters[j].a;
        int p2 = twinClusters[j].b;
        // Here starts the constraint
        ++noCtr;
        row_lb[noCtr] = -1*model->getInfinity();
        row_ub[noCtr] = 0;
        CoinPackedVector row;
        // Follow the variables Z related to the clusters p1 and p2 in the
list
        int index1 = calc_Index(t, p1, nClusters);
        row.insert(varZ[index1], 1);
        int index2 = calc_Index(t, p2, nClusters);
        row.insert(varZ[index2], 1);
        // The variable V corresponding to this pair of twins
        int index3 = calc_Index(t, j, twinClusters.size());
        row.insert(varV[index3], -2);
        // We search whether any of the two twins (p1 or p2) appear in
any other row
        for(unsigned jj=0; jj<twinClusters.size(); jj++)
                {
                if(jj != j)
                        {
                        if(twinClusters[jj].a == p1
                                || twinClusters[jj].b == p1
```

```
                                        || twinClusters[jj].a == p2
                                        || twinClusters[jj].b == p2)
                                        {
                                        int index = calc_Index(t, jj,
twinClusters.size());
                                        row.insert(varV[index], -2);
                                        }
                                }
                        }
                matrix->appendRow(row);
                }
        }
        // Constraint #5: Assign clusters to days
        for(int t=0; t<nDays; t++)
                {
                for(int j=0; j<nClusters; j++)
                        {
                        ++noCtr;
                        row_lb[noCtr] = -1*model->getInfinity();
                        row_ub[noCtr] = 0;
                        CoinPackedVector row;
                        for(int i=0; i<nCustomers; i++)
                                {
                                for(int r=0; r<nPatterns; r++)
                                        {
                                        int index1 = calc_Index(r, t, nDays);
                                        int index2 = calc_Index(r, i, nCustomers);
                                        int index3 = calc_Index(i, j, nClusters);
                                        if(M[index1] && O[index2] && L[index3])
                                                {
```

```
                              int index = calc_Index(i, r, nPatterns);
                              row.insert(varX[index], 1);
                              }
                         }
                    }
               int index = calc_Index(t, j, nClusters);
               row.insert(varZ[index], -1*bigM);
               matrix->appendRow(row);
               }
          }
     model->loadProblem(*matrix, col_lb, col_ub, objective, row_lb, row_ub);
     fLog << "Number of variables: " << model->getNumCols() << endl
          <<  "Number of constraints: " << model->getNumRows() << endl
          << "Number of variables: " << model->getNumElements() << endl;
     model->setHintParam(OsiDoReducePrint,true, OsiHintDo);
     for(int i=0; i<nCustomers; i++)
          {
          for(int r=0; r<nPatterns; r++)
               {
               int index = calc_Index(r, i, nCustomers);
               if(O[index])
                    {
                    int index = calc_Index(i, r, nPatterns);
                    model->setInteger(varX[index]);
                    }
               }
          }
     for(int t=0; t<nDays; t++)
          {
          for(int j=0; j<nClusters; j++)
```

```
				{
				int index = calc_Index(t, j, nClusters);
				model->setInteger(varZ[index]);
				}
			}
		for(int t=0; t<nDays; t++)
			{
			for(unsigned j=0; j<twinClusters.size(); j++)
				{
				int index = calc_Index(t, j, twinClusters.size());
				model->setInteger(varV[index]);
				}
			}
		CbcModel BB(*model);
		CbcHeuristicFPump *heur1 = new CbcHeuristicFPump();
		CbcRounding *heur2 = new CbcRounding();
		BB.addHeuristic(heur1);
		BB.addHeuristic(heur2);
		if(BB.getNumElements() < 0.3*(noCtr+1)*(noVar+1))
			BB.solver()->setHintParam(OsiDoPresolveInInitial,true);
		else
			BB.solver()->setHintParam(OsiDoPresolveInInitial,false);
		BB.setMaximumSeconds(120);
		BB.setLogLevel(0);
		BB.setAllowableGap(nDays*15);
		BB.setAllowableFractionGap(1);
		BB.branchAndBound();
		//if(k==0 && successfulAssignment==1)
		//	fLog << "#Visits(avg=" << (int) vehicle[k].Vmax/nDays  << "): ";
		if(BB.isProvenOptimal())
```

```
{
successfulAssignment = 1;
nOverlaps += (int) BB.getObjValue();
const double *solution = BB.bestSolution();
int maxOT= 0;
for(int t=0; t<nDays; t++)
	{
	int var = varUover[t];
	if((int) solution[var] > maxOT)
		maxOT = (int) solution[var];
	}
maxOT += (int) vehicle[k].Vmax/nDays;
maxOvertime.push_back(maxOT);
for(int t=0; t<nDays; t++)
	{
		fout << "D" << t+1 << "V" << k+1 << " ";
	for(int r=0; r<nPatterns; r++)
		{
		int index = calc_Index(r, t, nDays);
		if(M[index])
			{
			for(int i=0; i<nCustomers; i++)
				{
				int index = calc_Index(r, i, nCustomers);
				if(O[index])
					{
					int index = calc_Index(i, r, nPatterns);
					int var = varX[index];
					if(solution[var] > 0.1)
```

```
                                                            {
                                                            fout << translate[i] << " ";
                                                            for(unsigned ii=0;
ii<customer.size(); ii++)
                                                                  if(customer[ii].id
== translate[i])
                                                                        {
                                                                        break;
                                                                        }

                                                            }
                                                      }
                                                }
                                          }
                                    }
                              fout << endl;
                              }
                        }
                  else if(BB.isProvenInfeasible())
                        successfulAssignment = 0;
                  else
                        {
                        // time is up
                        successfulAssignment = 0;
                        }
                  delete heur1;
                  delete model;
                  delete matrix;
                  delete [] varX;
                  delete [] varZ;
                  delete [] varUunder;
```

```
		delete [] varUover;
		delete [] varV;
		delete [] objective;
		delete [] col_lb;
		delete [] col_ub;
		delete [] row_lb;
		delete [] row_ub;
		delete [] ctrDayVisits;
		if(successfulAssignment == 1)
			{
				fLog << "Max number of visits per day: " << maxOvertime[k]+1
<< endl;
			if(maxOvertime[k] >= 1.5*(int) vehicle[k].Vmax/nDays)
				successfulAssignment = 0;
			}
		fLog << "Assigning customers to days for vehicle " << k+1 << " lasted " <<
(double)(clock() - tVeh)/CLOCKS_PER_SEC << "sec";
		if(!BB.isProvenOptimal())
			fLog << " but will start again either because problem was infeasible or
time is up (more than 2min)";
		fLog << endl;
		if(successfulAssignment == 0)
			{
			k--;
			nFailures++;
			if(nFailures == 5)
				successfulAssignment = -1;
			}
		if(successfulAssignment == -1)
			break;
		}
```

```
        return successfulAssignment;
        }


void kmeans(vector<Customer> &customer, vector<Customer> &CustomersAssignedToVehicle, vector <TwoNumbers<int> > &twinClusters, int k, int nVehicles, int nDays, int  nClusters)
{
        int nkCustomers = CustomersAssignedToVehicle.size();
        // Clustering of the customers per vehicle using k-means k=5
        bool kmeansConverges = false; // k-means will not converge if it will take it more than 100 iterations
        while(!kmeansConverges)
        {
                if(twinClusters.size())
                        twinClusters.erase(twinClusters.begin(),twinClusters.end());
                // This 2D-vector will contain all i customers assigned to every j cluster of this vehicle
                vector < vector<Customer> > CustomersAssignedToClusters;
                int nVisits = 0;
                for(int i=0; i<nkCustomers; i++)
                        nVisits +=  CustomersAssignedToVehicle[i].f;
                int minAcceptedSizeOfCluster = 2*nVisits/nDays; //min(2*nVisits/nDays, nkCustomers/nClusters);
                // kmeans starts here
                // 1. Initially find k random points = seeds to start with


                // This vector will include the centroids (id, lat, long) of all clusters of this vehicle
                vector<Centre> CentroidsOfClusters;
                for(int j=0; j<nClusters; j++)
                {
                        int seed = rand()%nkCustomers;
```

```
                // Start with a seed, but make sure you do not select an already selected
one
                while(std::find_if(CentroidsOfClusters.begin(),
CentroidsOfClusters.end(),       MatchesID(CustomersAssignedToVehicle[seed].id))       !=
CentroidsOfClusters.end())
                        seed = rand()%nkCustomers;
                Centre     aCentroid     =     {CustomersAssignedToVehicle[seed].id,
CustomersAssignedToVehicle[seed].x, CustomersAssignedToVehicle[seed].y, 0, 0, 0, 0};
                CentroidsOfClusters.push_back(aCentroid);
        }
        // The "stop" criterion will turn true either when algorithm does not converge
        // after 100 iterations or when it has converged
        bool stop = false;
        // We keep track of the iterations. If more than 100, then we stop and repeat
        // the procedure with another set of seeds
        int nIterations = 0;
        while(!stop)
        {
                // Assign customers to clusters based on their distance from centroids
                for(int  i=0;  i<nkCustomers;  i++)  //  Memo:   nkCustomers  is
CustomersAssignedToVehicle.size();
                {
                        // This is the closest_cluster to the customer
                        int closest_cluster;
                        double min = 10e10;
                        for(int j=0; j<nClusters; j++)
                        {
                                double  dist  =  sqrt((CustomersAssignedToVehicle[i].x-
CentroidsOfClusters[j].x)*(CustomersAssignedToVehicle[i].x-CentroidsOfClusters[j].x)
                                        +(CustomersAssignedToVehicle[i].y-
CentroidsOfClusters[j].y)*(CustomersAssignedToVehicle[i].y-CentroidsOfClusters[j].y));
                                if(dist < min)
```

```
                        {
                                min = dist;
                                closest_cluster = j;
                        }
                }
                // This customer is assigned to closest_cluster (this may change
in the next iteration)
                CustomersAssignedToVehicle[i].clusteredTo = closest_cluster;
        }
        // Calculate new centroids
        int votesToStop = 0;
        for(int j=0; j<nClusters; j++)
        {
                // This is a temporary cluster created to be appended to
                // the 2D vector CustomersAssignedToClusters
                vector<Customer> aCluster;
                double temp_x = 0;
                double temp_y = 0;
                for(int i=0; i<nkCustomers; i++)
                {
                        if(CustomersAssignedToVehicle[i].clusteredTo == j)
                        {

    aCluster.push_back(CustomersAssignedToVehicle[i]);
                                temp_x += CustomersAssignedToVehicle[i].x;
                                temp_y += CustomersAssignedToVehicle[i].y;
                        }
                }
                // Vote to stop: One vote is cast
                if(abs(CentroidsOfClusters[j].x - temp_x/aCluster.size())<0.001
&& abs(CentroidsOfClusters[j].y - temp_y/aCluster.size())<0.001)
```

```
                    votesToStop++;
                CentroidsOfClusters[j].x = temp_x/aCluster.size();
                CentroidsOfClusters[j].y = temp_y/aCluster.size();
                if(aCluster.size())
                    CustomersAssignedToClusters.push_back(aCluster);
            }
            if(votesToStop == nClusters)
                stop = true;
            if(++nIterations > 100)
            {
                // k-means did not converge (usually at more than 100 iterations, smt is wrong)
                kmeansConverges = false;
                // Stop and restart...
                stop = true;
                // ... clustering for the same k-vehicle
                k--;
            }
            else
                kmeansConverges = true;
        }
        // if kmeans converges means that we have nClusters for the k-vehicle
        // that will not change. So I can now updated everything
        if(kmeansConverges)
        {
            // Update the struct customers to include clusters
            for(int i=0; i<nkCustomers; i++)
            {
                int ii = CustomersAssignedToVehicle[i].iter;
                customer[ii].clusteredTo = CustomersAssignedToVehicle[i].clusteredTo;
```

```
}
// Build twinClusters matrix:
// If a cluster is too small => nVisits <= Vmax/nDays,
// then it needs to be twinned with a close cluster
// Later on, we will allow overlaps between these two clusters
vector <int> visitsAtCluster;
// This vector will show whether cluster j has been twinned already or
not
vector <int> isTwinned;
for(int j1=0; j1<nClusters; j1++)
{
	// If a cluster is small...
	if((int)	CustomersAssignedToClusters[j1].size()	<
minAcceptedSizeOfCluster
		&& std::find(isTwinned.begin(), isTwinned.end(), j1) ==
isTwinned.end())
	{
		isTwinned.push_back(j1);
		int	combinedVisits	=
CustomersAssignedToClusters[j1].size();
		while(combinedVisits < minAcceptedSizeOfCluster)
		{
			// .. then find the closest cluster to this one
			double min = 10e10;
			int closest_cluster;
			for(int j2=0; j2<nClusters; j2++)
			{
				if(std::find(isTwinned.begin(),
isTwinned.end(), j2) == isTwinned.end())
				{
```

```
                                        double dist = sqrt((CentroidsOfClusters[j1].x-CentroidsOfClusters[j2].x)*(CentroidsOfClusters[j1].x-CentroidsOfClusters[j2].x)
    +(CentroidsOfClusters[j1].y-CentroidsOfClusters[j2].y)*(CentroidsOfClusters[j1].y-CentroidsOfClusters[j2].y));
                                        if(dist < min)
                                        {
                                            min = dist;
                                            closest_cluster = j2;
                                        }
                                    }
                                }
                                combinedVisits += CustomersAssignedToClusters[closest_cluster].size();
                                TwoNumbers<int> twoClusters = {j1, closest_cluster};
                                twinClusters.push_back(twoClusters);
                                isTwinned.push_back(closest_cluster);
                            }
                        }
                    }
                }
            }
}



void performAnalysis(vector<Customer> &customer, Centre barycentre, ofstream &fLog)
{
    double avgTurnover = 0;
    int nCustomers = customer.size();
```

```
for(int i=0; i<nCustomers; i++)
        avgTurnover += customer[i].q;
avgTurnover /= nCustomers;
double stdvTurnover = 0;
for(int i=0; i<nCustomers; i++)
        stdvTurnover += (customer[i].q - avgTurnover)*(customer[i].q - avgTurnover);
stdvTurnover = sqrt(stdvTurnover/nCustomers);
vector<Customer> suspiciousCustTurnover;
for(int i=0; i<nCustomers; i++)
{
        if(customer[i].q > avgTurnover+2*stdvTurnover)
                suspiciousCustTurnover.push_back(customer[i]);
}
int nSuspCustTO = suspiciousCustTurnover.size();
fLog << "While the average value of orders is " << avgTurnover;
if(stdvTurnover/avgTurnover > 1)
        fLog << " (albeit highly volatile) ";
fLog << "the following customers have extremely higher value of order" << endl;
std::sort(suspiciousCustTurnover.begin(),                suspiciousCustTurnover.end(),
higherTurnover);
for(int i=0; i<nSuspCustTO; i++)
        fLog << "Customer ID: " << suspiciousCustTurnover[i].id << " Turnover: " <<
suspiciousCustTurnover[i].q << endl;


double avgDist = 0;
for(int i=0; i<nCustomers; i++)
{
        double dist = sqrt((customer[i].x-barycentre.x)*(customer[i].x-barycentre.x) +
                (customer[i].y-barycentre.y)*(customer[i].y-barycentre.y));
        avgDist += dist;
}
```

```
        avgDist /= nCustomers;
        double stdvDist = 0;
        vector<TwoNumbers<double>> custDist;
        for(int i=0; i<nCustomers; i++)
        {
                double dist = sqrt((customer[i].x-barycentre.x)*(customer[i].x-barycentre.x) +
                        (customer[i].y-barycentre.y)*(customer[i].y-barycentre.y));
                stdvDist += (dist-avgDist)*(dist-avgDist);
                TwoNumbers<double> aCustDist= {customer[i].id, dist};
                custDist.push_back(aCustDist);
        }
        stdvDist = sqrt(stdvDist/nCustomers);
        vector<TwoNumbers<double>> suspiciousCustDist;
        for(int i=0; i<nCustomers; i++)
        {
                if(custDist[i].b > avgDist+2*stdvDist)
                        suspiciousCustDist.push_back(custDist[i]);
        }
        int nSuspCustDist = suspiciousCustDist.size();
        fLog << "The following customers are far away from the centre of all points" << endl;
        for(int i=0; i<nSuspCustTO; i++)
                fLog << "Customer ID: " << suspiciousCustDist[i].a << endl;
}
void radarScan(vector<Customer> &customer, vector<Vehicle> &vehicle, Centre &barycentre, ofstream &fLog)
{
        fLog << "Cluster nodes to vehicles with Radar Scan" << endl;
        int nCustomers = customer.size();
        int nVehicles = vehicle.size();
        // Calculate angles phi
```

```
double min = 10e10;
int bestSeed;
int nVisits = 0;
for(int i=0; i<nCustomers; i++)
        nVisits += customer[i].f;
for(int seed=0; seed<nCustomers; seed++)
{
        std::sort(customer.begin(), customer.end(), asReadInFile);
        Centre p = {0, customer[seed].x, customer[seed].y, 0, 0};
        double zx0 = p.x - barycentre.x;
        double zy0 = p.y - barycentre.y;
        for(int i=0; i<nCustomers; i++)
        {
                double zxi = customer[i].x - barycentre.x;
                double zyi = customer[i].y - barycentre.y;
                double dot = zxi*zx0 + zyi*zy0;
                double pcross = -zx0*zyi + zxi*zy0;
                customer[i].phi = atan2(pcross, dot)*180/3.1415;
        }

        // Clustering following phi angle
        for(int k=0; k<nVehicles; k++)
        {
                vehicle[k].Q = 0;
                vehicle[k].V = 0;
        }
        // Sort customers based on their angle
        std::sort(customer.begin(), customer.end(), smallerPhi);
        // kIsOpen means that the k-centroid is not yet full
        // When centroid becomes full, isOpen will turn to false
```

```
// At first, all centroids are open
bool *kIsOpen = new bool [nVehicles];
for(int k=0; k<nVehicles; k++)
        kIsOpen[k] = true;
// The following will be used because I want to calculate centroids
// and distances at each cluster. I need the x and y cooridnates
// and the size of the customers assigned to cluster
double *x = new double [nVehicles];
double *y = new double [nVehicles];
int *size = new int [nVehicles];
for(int k=0; k<nVehicles; k++)
{
        x[k] = 0;
        y[k] = 0;
        size[k] = 0;
}
// Customers are sorted based on their phi angle from barycentre
double startPhi = customer[seed].phi;
for(int i=0; i<nCustomers; i++)
{
        for(int k=0; k<nVehicles; k++)
        {
                // Customer i will be assigned to the first available k
                if(vehicle[k].V + customer[i].f < nVisits/nVehicles
                        && kIsOpen[k])
                {
                        customer[i].assignedTo = k;
                        vehicle[k].V += customer[i].f;
                        vehicle[k].Q += customer[i].f*customer[i].q;
                        x[k] += customer[i].x;
```

```
                y[k] += customer[i].y;
                size[k]++;
                break;
            }
            // But if there is no avaiable k, then the customer i will
            // be assigned to the last vehicle, even if it is full
            else if(k == nVehicles-1)
            {
                customer[i].assignedTo = k;
                vehicle[k].V += customer[i].f;
                vehicle[k].Q += customer[i].f*customer[i].q;
                x[k] += customer[i].x;
                y[k] += customer[i].y;
            }
            // If customer i cannot be assigned to k (which is still open)
because
            // it just overpassed its capacity, then it's time to close k.
            else if(kIsOpen[k])
            {
                kIsOpen[k] = false;
                x[k] /= size[k];
                y[k] /= size[k];
                continue;
            }
        }
    }

    // Last vehicles's x and y were not divided by size
    x[nVehicles-1] /= size[nVehicles-1];
    y[nVehicles-1] /= size[nVehicles-1];
```

```
            // Calculate distances and coefficient of variation = stdv/avg and choose
update min
            double minDist = 10e10;
            double maxDist = 0;
            for(int k=0; k<nVehicles; k++)
            {
                  double dist = sqrt((x[k]-barycentre.x)*(x[k]-barycentre.x) +
                        (y[k]-barycentre.y)*(y[k]-barycentre.y));
                  if(dist < minDist)
                        minDist = dist;
                  if(dist > maxDist)
                        maxDist = dist;
            }
            if(maxDist - minDist < min)
            {
                  min = maxDist - minDist;
                  bestSeed = seed;
            }
            delete [] kIsOpen;
            delete [] x;
            delete [] y;
            delete [] size;
      }
      // I repeat the process only for the seed I selected as bestSeed
      Centre p = {0, customer[bestSeed].x, customer[bestSeed].y, 0, 0};
      double zx0 = p.x - barycentre.x;
      double zy0 = p.y - barycentre.y;
      for(int i=0; i<nCustomers; i++)
      {
            double zxi = customer[i].x - barycentre.x;
```

```
        double zyi = customer[i].y - barycentre.y;
        double dot = zxi*zx0 + zyi*zy0;
        double pcross = -zx0*zyi + zxi*zy0;
        customer[i].phi = atan2(pcross, dot)*180/3.1415;
}

// Clustering following phi angle
for(int k=0; k<nVehicles; k++)
{
        vehicle[k].Q = 0;
        vehicle[k].V = 0;
}
// Sort customers based on their angle
std::sort(customer.begin(), customer.end(), smallerPhi);
// kIsOpen means that the k-centroid is not yet full
// When centroid becomes full, isOpen will turn to false
// At first, all centroids are open
bool *kIsOpen = new bool [nVehicles];
for(int k=0; k<nVehicles; k++)
        kIsOpen[k] = true;
// I DONT USE BORDERS THROUGHOUT THE CODE.
// BUT I INCLUDE CODE HERE TO CALCULATE BORDERS
// IN CASE OF FUTURE REFINEMENT
vector<TwoNumbers<int>> borders;
int start = 0;
// Customers are sorted based on their phi angle from barycentre
for(int i=0; i<nCustomers; i++)
{
        for(int k=0; k<nVehicles; k++)
        {
```

```
                // Customer i will be assigned to the first available k
                if(vehicle[k].V + customer[i].f < nVisits/nVehicles
                        && kIsOpen[k])
                {
                        customer[i].assignedTo = k;
                        vehicle[k].V += customer[i].f;
                        vehicle[k].Q += customer[i].f*customer[i].q;
                        break;
                }
                // But if there is no avaiable k, then the customer i will
                // be assigned to the last vehicle, even if it is full
                else if(k == nVehicles-1)
                {
                        customer[i].assignedTo = k;
                        vehicle[k].V += customer[i].f;
                        vehicle[k].Q += customer[i].f*customer[i].q;
                }
                // If customer i cannot be assigned to k (which is still open) because
                // it just overpassed its capacity, then it's time to close k.
                else if(kIsOpen[k])
                {
                        TwoNumbers<int> startEnd = {start, i-1};
                        borders.push_back(startEnd);
                        start = i;
                        kIsOpen[k] = false;
                        continue;
                }
        }
}
TwoNumbers<int> startEnd = {start, nCustomers-1};
```

```
        borders.push_back(startEnd);
        delete [] kIsOpen;
        // I will check which customers are close to borders
        // Angle phi will help distinghuish them
        bool ** isCloseToBorder = new bool *[nCustomers];
        for(int i=0; i<nCustomers; i++)
                isCloseToBorder[i] = new bool [nVehicles];
        for(int i=0; i<nCustomers; i++)
                for(int k=0; k<nVehicles; k++)
                        isCloseToBorder[i][k] = false;
        double win = nCustomers/nVehicles*0.05;
        for(int k=0; k<nVehicles; k++)
        {
                int ii = borders[k].a;
                if(ii != 0)
                {
                        for(int i=ii-(int) win/2; i<ii; i++)
                                isCloseToBorder[customer[i].iter][k] = true;
                        for(int i=ii; i<ii+(int) win/2; i++)
                                isCloseToBorder[customer[i].iter][k-1] = true;
                }
                else
                {
                        for(int i=nCustomers-(int) win/2; i<nCustomers; i++)
                                isCloseToBorder[customer[i].iter][0] = true;
                        for(int i=ii; i<ii+(int) win/2; i++)
                                isCloseToBorder[customer[i].iter][nVehicles-1] = true;
                }
        }
}
```

```
int assignCustomersToVehicles(vector<Customer> &customer, vector<Vehicle> &vehicle,
double varVisits, double varTurnover, vector<FromTo<bool>> &ft, ofstream &fLog)
{
	fLog << endl << "Assignment of customers to vehicles" << endl;
	int nCustomers = customer.size();
	int nVehicles = vehicle.size();
	double Q = 0;
	int nVisits = 0;
	for(int i=0; i<nCustomers; i++)
	{
		Q += customer[i].q*customer[i].f;
		nVisits += customer[i].f;
	}
	// COIN-OR STARTS HERE.
	// We will use the Osi Interface to create the problem.
	// Osi is an interface that could be used to pass the
	// created problem to many solvers. When we create an Osi
	// interface we then define the OsiXXXSolverInterface
	// where XXX may be Clp (a COIN LP solver), GLPK, CPLEX
	// or whichever other solver one can use

	// noVar will be the iterator of the number of variables
	// we have added in the problem. It starts at -1 and will
	// turn 0 when the first variable is defined.
	int noVar = -1;
	// varX will store the variables. In COIN-OR the
	// variables are referenced by a single integer: 0, 1, 2 etc
	// So when we pass say the 12th variable, we want to remember
	// that this variable represents e.g. the variable y(4).
```

```
// So we need to have y(4) = 12 and we will refer to this
// variable not as 12, but as y[4]. This is what varY does
int *varX;
// Here we define the coefficients in the objective function
// per variable, the lower/upper bounds for each variable,
// and the lower/upper bounds of the constraints.
double *objective, *col_lb, *col_ub, *row_lb, *row_ub;
// The number of variables-columns and constraints-rows can be
// the actual number or a larger one.
int nCols = nCustomers*nVehicles;
int nRows = nCustomers+nVehicles+nVehicles;
// We allocate memory for each one
varX       = new int    [nCols];
objective  = new double [nCols];
col_lb     = new double [nCols];
col_ub     = new double [nCols];
row_lb     = new double [nRows];
row_ub     = new double [nRows];
// Create an OSI interface called model
// Since we know we will solve the problem using the
// Clp solver we instantiate the object model as
// an OsiClp solver interface - we could also pas Cpx for CPLEX
OsiSolverInterface *model = new OsiClpSolverInterface();

// Next we define the variables. That is, we correspond the first
// column noVar=0 to the variable varX[0] and so on.
// varX represents the variable x(i,j), so it takes two indices:
// i and j. COIN works better with one-dimension arrays. So we
// surely want to avoid manipulation of 2D matrices. For this reason
// we have defined a simple function calc_Index which takes the
```

```
// two indices i and j, and based on the number of columns N, it
// returns a single index: i*N+j which in fact maps the
// position of the item (i,j) onto a single dimension.
// Step by step: we calculate the index using (i,j). We assign the variable
// to this index. We fix the lower and upper bound of the variable
// and define the coefficient in the objective function.
for(int i=0; i<nCustomers; i++)
{
        for(int k=0; k<nVehicles; k++)
        {
                int index = calc_Index(customer[i].iter, k, nVehicles);
                varX[index] = ++noVar;
                col_lb[noVar] = 0;
                col_ub[noVar] = 1;
                if(customer[i].assignedTo == k)
                        objective[noVar] = 0;
                //else if(isCloseToBorder[customer[i].iter][k])
                //      objective[noVar] = 0;
                else
                        objective[noVar] = 1;
        }
}

// Up to this time, we have passed no information yet to our model.
// The creation of the variables served only for the next phase.
// Up to now the model is empty!

// We now define the technological matrix of the constraints.
// The constraints will be created based on the variables that we
// created in the previous step and their coefficients in the
```

```
// technological matrix.
CoinPackedMatrix * matrix = new CoinPackedMatrix(false, 0,0);
matrix->setDimensions(0, noVar+1);
// noCtr is like noVar. It will be the iterator of the number
// of constraints we have added in the problem. It starts at
// -1 and will turn 0 when the first constraint is defined.
int noCtr = -1;
// Constraints are similar to variables. In the future, when we
// wish to obtain the dual values of reduced costs of this
// constraint instead of asking getDualValue(20) we will ask
// getDualValue(conCustomer[20]).
int *conAssign   = new int [nCustomers];
int *conTurnover = new int [nVehicles];
int *conVisits = new int [nVehicles];
// Next we define each row of the constraint
// Step by step: we assign the current iterator to the constraint
// we want to recall later. We define lower and upper bounds.
// And then we proceed to creating a row
// vector that will contain for every variable its coefficient. We insert
// this product x(i,j)*a(i,j) in the row using the row.insert method.
// When all elements of the row/constraint are added, we append this row
// in the technological matrix using matrix->appendRow(row).
// NOTE: We use "->" when the object (e.g. "matrix") is a pointer and "." when
// the object (e.g. "row") is not a pointer.

// Assignment
for(int i=0; i<nCustomers; i++)
{
        conAssign[i] = ++noCtr;
        row_lb[noCtr] = 1;
```

```
        row_ub[noCtr] = 1;
        CoinPackedVector row;
        for(int k=0; k<nVehicles; k++)
        {
                int index = calc_Index(customer[i].iter, k, nVehicles);
                row.insert(varX[index], 1);
        }
        matrix->appendRow(row);
}
// Turnover
for(int k=0; k<nVehicles; k++)
{
        conTurnover[k] = ++noCtr;
        row_lb[noCtr] = varTurnover*Q/nVehicles;
        row_ub[noCtr] = (2-varTurnover)*Q/nVehicles;
        CoinPackedVector row;
        for(int i=0; i<nCustomers; i++)
        {
                int index = calc_Index(customer[i].iter, k, nVehicles);
                row.insert(varX[index], customer[i].q*customer[i].f);
        }
        matrix->appendRow(row);
}
// Number of visits per vehicle
for(int k=0; k<nVehicles; k++)
{
        conVisits[k] = ++noCtr;
        row_lb[noCtr] = varVisits*nVisits/nVehicles;
        row_ub[noCtr] = (2-varVisits)*nVisits/nVehicles;
        CoinPackedVector row;
```

```
        for(int i=0; i<nCustomers; i++)
        {
                int index = calc_Index(customer[i].iter, k, nVehicles);
                row.insert(varX[index], customer[i].f);
        }
        matrix->appendRow(row);
}
// Load problem to OSI model
// We have almost all we need to load our model: we have variables,
// bounds for these variables, constraints and bounds, and coefficients.
// We do not know yet which variables are integer if any. All are float.
model->loadProblem(*matrix, col_lb, col_ub, objective, row_lb, row_ub);
fLog << "Number of variables: " << model->getNumCols() << endl
        <<  "Number of constraints: " << model->getNumRows() << endl
        << "Number of variables: " << model->getNumElements() << endl;

// This would solve the relaxation of the problem
// Reduce verbosity of the solver. If put at false, too much
// info will flood the screen
model->setHintParam(OsiDoReducePrint,true, OsiHintDo);



for(int i=0; i<nCustomers; i++)
{
        for(int k=0; k<nVehicles; k++)
        {
                int index = calc_Index(customer[i].iter, k, nVehicles);
                model->setInteger(varX[index]);
        }
}
```

```
CbcModel BB(*model);
CbcHeuristicFPump *heur1 = new CbcHeuristicFPump();
heur1->setMaximumPasses(100);
BB.addHeuristic(heur1);
BB.setLogLevel(0);
BB.setAllowableFractionGap(1);
BB.branchAndBound();
delete heur1;
// First initialize vehicle current visits and Q just to be safe
// Will later update them
for(int k=0; k<nVehicles; k++)
{
        vehicle[k].Q = 0;
        vehicle[k].V = 0;
}
// Check whether there exist cases where x(i,k) = 1 but the objective
// coeeficient is 1, that is, check when a customer is NOT assigned
// to the indended vehicle and mark where this customer is assigned
// and where he should have been assigned
// This is called the ft table (from-to): A vehicle requests a move
// from "from" to  "to".
if(BB.isProvenOptimal())
{
        const double *solution = BB.bestSolution();
        for(int k=0; k<nVehicles; k++)
        {
                for(int i=0; i<nCustomers; i++)
                {
                        int index = calc_Index(customer[i].iter, k, nVehicles);
                        int var = varX[index];
```

```
if(solution[var] > 10e-3) // 10e-3 is only for numerical purposes. Means essentially "not 0"
{
    customer[i].assignedTo = k;
    // Update Q (current turnover) and V (current num of visits) for each vehicle
    int thisCustomer = customer[i].iter;
    vehicle[k].Q += customer[thisCustomer].q*customer[thisCustomer].f;
    vehicle[k].V += customer[thisCustomer].f;
    // Check to see whether this customer should have been assigned elsewhere
    if(objective[var] == 1)
    {
        // The customer should have been assigned toThisVehcle
        int toThisVehicle;
        for(int kk=0; kk<nVehicles; kk++)
        {
            int index = calc_Index(customer[i].iter, kk, nVehicles);
            int var = varX[index];
            if(objective[var] == 0)
                toThisVehicle = kk;
        }
        FromTo<bool> aRequest = {customer[i].iter, k, toThisVehicle, false};
        ft.push_back(aRequest);
    }
}
}
}
```

```
        }
        else
        {
                fLog << "Assignment of customers to vehicles not optimal. Exit. " << endl;
                return 1;
        }
        int nOverlaps = (int) BB.getObjValue();
        if(BB.getObjValue()-0.1 > nOverlaps) // -0.1 is only put for numerical purposes
                nOverlaps++;
        delete model;
        delete matrix;
        delete [] varX;
        delete [] objective;
        delete [] col_lb;
        delete [] col_ub;
        delete [] row_lb;
        delete [] row_ub;
        delete [] conAssign;
        delete [] conTurnover;
        delete [] conVisits;
return nOverlaps;
}
```