

JOpt.ASP – Developers Guide

Inhaltsverzeichnis

JOpt.ASP – Developers Guide.....	1
Creating a Client application.....	1
Hello JOpt.....	2
Adding an optimization progress indicator.....	5
Sample Progress reporter Class:.....	6
Usage.....	7
Using time and distance matrices.....	8
Using optimization properties.....	10
Properties that affect the overall optimization behaviour.....	10
Properties that affect the optimization result according to multiple objectives.....	10
Carbon Dioxide Assessment and Optimization.....	12
Calculation scheme.....	12
Fuel Usage and Carbon Emission.....	13
Load Distance.....	14
Energy Efficiency.....	15
Fuel & Emission per Load Kilometer.....	15
Obtaining the results.....	16
What's next ?.....	17

Creating a Client application

Get Ready

- Open VisualStudio and create a new Project
- Create a new web reference for the JOpt service.
 - In the project tree right click on the webreference entry and press “add...”
- Enter the URL where the JOpt service is located
 - (e.g. http://localhost/aspnet/asp_net.asmx?wsdl)
- As soon as VisualStudio has obtained the service interface you will have to define a name for the service (e.g. joptaas).
- Press ok
- VisualStudio will now create all required classes and proxy interfaces that are required to access the service within your program.

Hello JOpt

First we have to create the primary optimization instance. While most common webservises are stateless, JOpt.ASP offers its functions via stateful webservises and therefore requires a client site cookie container in order to persue its state.

```
jopt_aspClass optimizer = new jopt_aspClass();  
CookieContainer cc = new CookieContainer();  
optimizer.CookieContainer = cc;
```

Further you will have to make some adjustments regarding the session timeout in order not to disrupt the current session until the service has returned its result:

```
optimizer.Timeout = 24 * 3600 * 1000; // 24 hours
```

Now we are well prepared to set up the optimization problem itself. We start with adding the resources. Before we can add a resource we will first have to define the resources working hours. Each resource can have multiple working hours of course, but for simplicity we only define one single working day for now:

```
JOptAspWorkingHours[] workingHours = new JOptAspWorkingHours[1];  
workingHours[0] = new JOptAspWorkingHours();  
workingHours[0].begin = new DateTime(2007, 3, 6, 8, 0, 0);  
workingHours[0].end = new DateTime(2007, 3, 6, 17, 0, 0);
```

Now we can add the resource:

```
JOptAspCapacityResource resource01 = new JOptAspCapacityResource();  
resource01.resourceId = "Truck1";  
resource01.latitude = 50.1167;  
resource01.longitude = 7.68333;  
resource01.maxHours = 12.0;  
resource01.maxDistance = 1200.0;  
resource01.workingHours = workingHours;  
  
JOptAspCapacityResource resource02 = new JOptAspCapacityResource();  
resource02.resourceId = "Truck2";  
resource02.latitude = 50.1167;  
resource02.longitude = 7.68333;  
resource02.maxHours = 12.0;  
resource02.maxDistance = 1200.0;  
resource02.workingHours = workingHours;  
  
optimizer.addResource(resource01);  
optimizer.addResource(resource02);
```

Besides having dedicated working hours each resource has a name and a home location (depot) that is defined by its latitude and longitude. The depot is usually the starting and end

point for each route that is performed by this specific resource (later on we will also discuss alternate route types like open routes and multi day routes with overnight stays).

By defining a maximum route distance (maxDistance) and a maximum route time (maxHours) we can tell the optimizer that this specific resource shall not perform routes that exceed these values. Hence reducing these values will spread the remaining nodes to other resources and timeslots if available.

Once having added the resources we can now add the nodes that have to be visited. Again we first have to define the opening hours for each node. Opening hours define timewindows where the node may be visited:

```
JOptAspOpeningHours[] openingHours = new JOptAspOpeningHours[1];
openingHours [0] = new JOptAspOpeningHours();
openingHours [0].begin = new DateTime(2007, 3, 6, 8, 0, 0);
openingHours [0].end = new DateTime(2007, 3, 6, 17, 0, 0);
```

In order to setup the nodes we could define a little helper class in order to have a more compact code (this is not necessary for JOpt but makes it more readable)

```
//Helper Class
class basicNode
{
    public String name;
    public double lat;
    public double lon;

    public basicNode(String name, double lat, double lon){
        this.name = name;
        this.lat = lat;
        this.lon = lon;
    }
}
```

```
basicNode[] basicnode = new basicNode[11];
basicnode[0] = new basicNode("Koeln", 50.9333, 6.95);
basicnode[1] = new basicNode("Dueren", 50.8, 6.48333);
basicnode[2] = new basicNode("Wuppertal", 51.2667, 7.18333);
basicnode[3] = new basicNode("Oberhausen", 51.4667, 6.85);
basicnode[4] = new basicNode("Essen", 51.45, 7.01667);
basicnode[5] = new basicNode("Mannheim", 49.4883, 8.46472);
basicnode[6] = new basicNode("Heilbronn", 49.1403, 9.22);
basicnode[7] = new basicNode("Stuttgart", 48.7667, 9.18333);
basicnode[8] = new basicNode("Muenchen", 48.15, 11.5833);
basicnode[9] = new basicNode("Nuernberg", 49.4478, 11.0683);
basicnode[10] = new basicNode("Augsburg", 48.36214, 10.89411);
```

DNA

//evolutions

```
for (int i = 0; i < basicnode.GetLength(0); i++)
{
    JOptAspTimeWindowGeoNode node = new JOptAspTimeWindowGeoNode();
    node.nodeId = basicnode[i].name;
    node.latitude = basicnode[i].lat;
    node.longitude = basicnode[i].lon;
    node.openingHours = weeklyOpeningHours;
    node.visitDuration = 1200;
    node.priority = 1;

    // add the nodes to the optimizer
    optimizer.addNode(node);
}
```

Finally we can run the optimizer...

```
JOptAspOptimizationResult optresult = null;
try
{
    optresult = optimizer.startRemoteOptimizationRun();
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
    Console.Read();
}
```

... and obtain the resulting solution:

```
if (optresult != null)
{
    Console.WriteLine("-----");
    Console.WriteLine("Distance = " + optresult.totalDistance / 1000 + " km");
    Console.WriteLine("Total Time = " + optresult.timeTotal / 3600 + " h");
    Console.WriteLine("Tot Trip Time = " + optresult.timeTrip / 3600 + " h");
    Console.WriteLine("Tot Node Time = " + optresult.timeStop / 3600 + " h");
    Console.WriteLine("Tot Break Time = " + optresult.timeBreak / 3600 + " h");
    Console.WriteLine("Tot Idle Time = " + optresult.timeIdle / 3600 + " h");
    Console.WriteLine("-----");

    foreach (JOptAspRoute route in optresult.routes)
    {
        Console.WriteLine("-----");
        Console.WriteLine("Route: " + route.routeId);
        Console.WriteLine("Distance = " + route.totalDistance / 1000 + " km");
        Console.WriteLine("Overall Time = " + route.timeTotal / 3600 + " h");
        Console.WriteLine("Trip Time = " + route.timeTrip / 3600 + " h");
        Console.WriteLine("Node Time = " + route.timeStop / 3600 + " h");
        Console.WriteLine("Break Time = " + route.timeBreak / 3600 + " h");
        Console.WriteLine("Idle Time = " + route.timeIdle / 3600 + " h");
        foreach (JOptAspViolation violation in route.violations)
```

DNA

//evolutions

```
{
    Console.WriteLine("ConstraintViolation {0} : {1} : {2}",
        violation.category,violation.type,violation.value);
}
Console.WriteLine("");
int i = 0;
foreach (JOptAspScheduledNode node in route.nodes)
{
    Console.WriteLine("{0}.{1}: {2} arrival: {3}",
        route.routeId,i++, node.nodeId,node.arrivalTime);

    foreach (JOptAspViolation violation in node.violations)
    {
        Console.WriteLine("{0}: ConstraintViolation {1} : {2} : {3}",
            node.nodeId,
            violation.category,
            violation.type,
            violation.value);

        Console.WriteLine("");
    }
}
} else
{
    Console.WriteLine("Result = null ");
}
```

Congratulations you have now solved your first optimization problem using a webservice !
Now lets have a look at some more advanced practices like:

- **Optimization progress indication**
- **Use of Distance and Time Matrices**
- **Optimization properties**
- **Weekly vs. Daily Planning Strategies**
- **Force daily and weekly breaks**
- **Node Priorities**
- **Vehicle Costs**
- **Dealing with very long servicing times (e.g. multiple days at one node)**

Adding an optimization progress indicator

When it comes to large scaled problems optimization may take several minutes or even hours to complete its calculation. In order to inform you about the optimization progress JOpt.ASP offers a session based progress service, that returns the current optimization progress on request. In order to obtain the progress information you will have to perform three simple steps:

- Create a secondary optimization instance.

DNA

//evolutions

- Using this secondary instance call the `getProgress()` method and pass the `sessionid` of the primary optimizer.
- The `getProgress()` method will then return an object that contains
 - The current progress
 - The current overall distance
 - The current costfunction's value. (A mere abstract value, that can be used to compare the current solutions quality against previous iterations)

Sample Progress reporter Class:

```
class progressReporter
{
    private jopt_aspClass progressService;
    private String sessionId;
    private volatile bool stop = false;

    public progressReporter(jopt_aspClass progressService, String sessionId)
    {
        this.progressService = progressService;
        this.sessionId = sessionId;
    }

    public void doGetProgress()
    {
        while (!stop)
        {
            JOptAspProgress progress =
                progressService.getOptimizationProgress(this.sessionId);

            if (progress != null)
            {
                Console.WriteLine("Progress:{0} TotalDistance{1} Costfunction: {2}",
                    progress.curProgress,
                    progress.curDistance,
                    progress.curCost);
            } else {
                Console.WriteLine("getOptimizationProgress() = empty ");
            }

            System.Threading.Thread.Sleep(5000);
        }
    }

    public void stopGettingProgress()
    {
        stop = true;
    }
}
```

DNA

//evolutions

Usage

```
static void Main(string[] args)
{
    try
    {
        jopt_aspClass optimizer = new jopt_aspClass();
        jopt_aspClass progressService = new jopt_aspClass();
        ....

        progressReporter backgroundReporter = new
        progressReporter(progressService, sessionid);

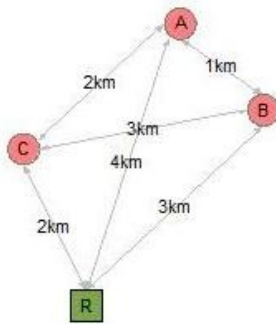
        Thread workerThread = new
            Thread(backgroundReporter.doGetProgress);

        workerThread.Start();
        ....

        JOptAspOptimizationResult optresult = null;
        try
        {
            optresult = optimizer.startRemoteOptimizationRun();
        }
    }
}
```

Using time and distance matrices

Without any special arrangements Jopt uses a very simple model of linear distances between the nodes: „as the crow flies“. If a distance matrix is omitted, this distance model will automatically be applied. For many applications this model is not good enough and should be exchanged with real values considering real driving distances and times. These distances are typically available as distance matrices. Such a matrix lists all distances a car or truck will need to go when driving from one node to another. The optimization algorithm needs all distances between each node so that a complete matrix is expected. Note, that there might be a difference between driving from A to B to driving from B to A, e.g. because of one-way roads. Therefore, a complete and non-symmetric matrix is necessary.



	A	B	C	R
A	0	1	2	4
B	1	0	3	3
C	2	3	0	2
R	4	3	2	0

```
//set up the matrix
double[][] matrix = {
{0.0,1.0,2.0,4.0},
{1.0,0.0,3.0,3.0},
{2.0,3.0,0.0,2.0},
{4.0,3.0,2.0,0.0}};

//add the nodes to be visited

TimeWindowGeoNode A = ...
A.setDistMatrixId(0);
this.addElement(A);

TimeWindowGeoNode B = ...
B.setDistMatrixId(1);
this.addElement(B);

TimeWindowGeoNode C = ...
C.setDistMatrixId(2);
this.addElement(C);
...
```

In this example all the distances between the nodes of are given as the variable double [][] matrix in kilometres. In order to enable rapid access to the matrix, you must allocate the proper matrix index as the node ID using the function setDistMatrixId(matrix_index). The matrix itself is added to the program by using addDistanceMatrix(matrix).

```
this.addDistanceMatrix(matrix);  
...
```

Once having defined the real driving distance you may also want to set up a time matrix. A time matrix defines the travel times between each pair of nodes in seconds.

```
double[][] timematrix = new double[4][4];  
for(int i = 0; i < timematrix.length; i++)  
{  
    for(int j = 0; j < timematrix[i].length; j++)  
    {  
        timematrix[i][j] = matrix[i][j] * 1000.0 / 22.0; // km *  
        1000 / speed [m/s] = time [s]  
    }  
}  
this.addTimeMatrix(timematrix);
```

Time matrices have been introduced in order to respect real travel times for different road types instead of linear calculated times by means of distance/ resources' average speed. Hence the resources' average speed will be ignored when having set the time matrix. However, you can define individual vehicle speeds by adding a vehicle specific time factor to the resource. In the example below resource R will travel twice as fast as given by the time matrix.

```
CapacityResource R = new  
CapacityResource("Res", 50.1167, 7.68333, 12.0, 1200.0, workingHours);  
R.setTimeMatrixFactor(0.5);  
R.setDistMatrixId(3);  
this.addElement(R);  
  
this.addDistanceMatrix(matrix);
```

Using optimization properties

Optimization properties can be used to influence the optimization general behaviour as well as affecting the optimization result according to multiple objectives. Like in real life, multiple objectives (optimum distance, time windows, load balance, etc.) will lead to a competitive condition between these objectives and one will have to make a trade-off decision and prioritise objectives.

Properties that affect the overall optimization behaviour

JOpt uses a so called genetic algorithm for optimisation. Genetic algorithms are basically a simulation of evolution in which a population of abstract representations of candidates to an optimization problem evolves toward better solutions with each iteration. During each optimization iteration JOpt permanently creates new solution candidates and propagates all promising candidates using genetic operators like mutation and recombination while deprecating all candidates with a bad fitness (survival of the fittest).

Property	Values	Description
JOptExitCondition.Type	JOptGenerationCount JOptConvergencyCount	exit condition
JOptExitCondition.Count	1..N	number of generations
JOptExitCondition.JOptConvergencyCount	1..N	number of generations
JOpt.RouteType	CLOSED OPEN	plan open or closed routes by default
JOpt.OptimizationRule	DAILY WEEKLY	automatically add overnight stays if required

Properties that affect the optimization result according to multiple objectives

The fitness of a solution is determined by a cost function where all the different – mostly concurring – optimization objectives are considered. In order to affect the solution towards a desired direction JOpt allows to prioritise different objectives by defining a weighting factor.

In the fitness function each objective is normalized to be comparable to a distance unit. Below you find the cost of each objective by units of kilometers. For example when having JoptWeight.RouteTime set to 1 each hour that exceeds the max time directive given for the respective resource will cost the same as 100 extra kilometer driving distance in the overall solution.

Property	Values	Description	Normalized
JOptWeight.TimeWindow	1..N	weight timewindow does not match (early/late)	100km/minute

DNA

//evolutions

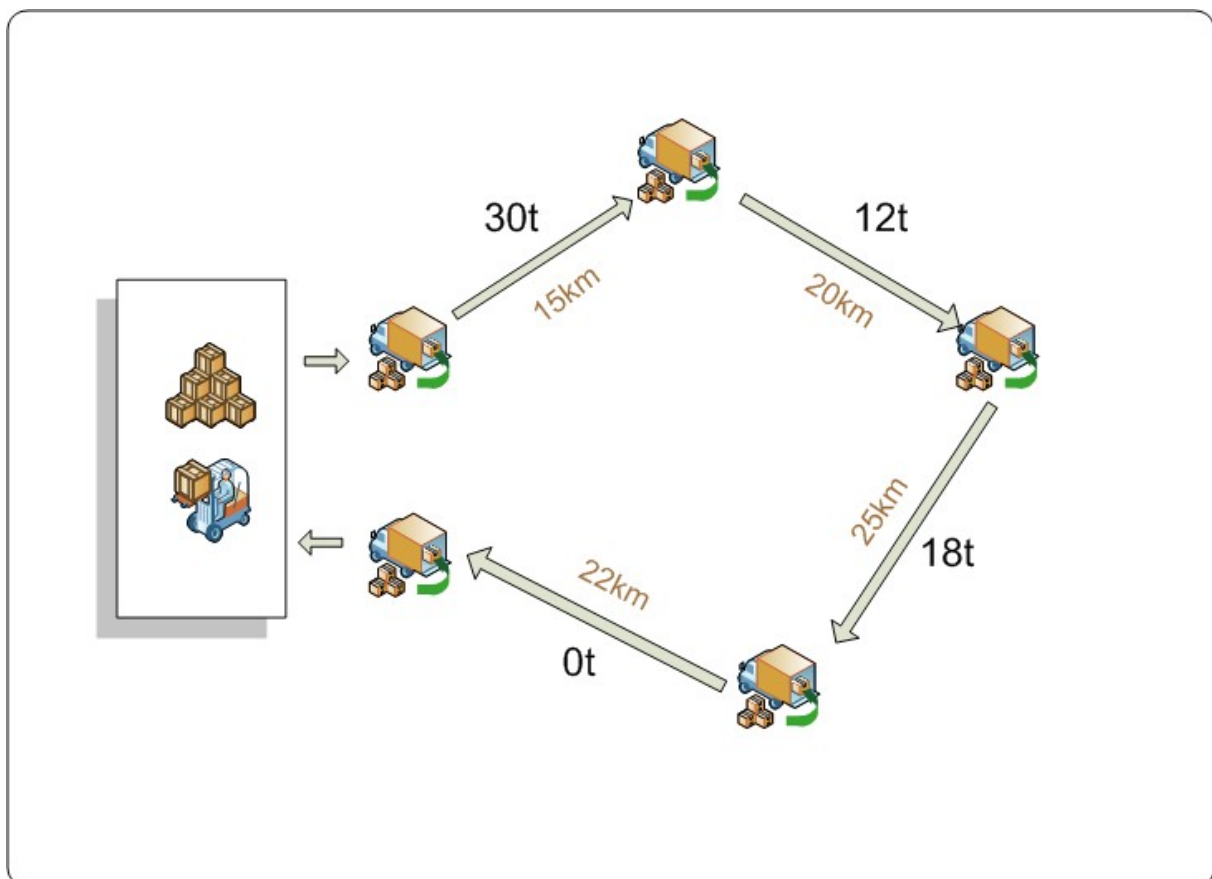
JOptWeight.TotalDistance	1..N	weight total distance	1km/km
JOptWeight.RouteDistance	1..N	weight max route distance exceed	10km/km
JOptWeight.RouteTime	1..N	weight max route time exceed	100km/hour
JOptWeight.Capacity	1..N	weight resource capacity exceed	1km/unit
JOptWeight.NodeType	1..N	weight node type permission does not match the vehicle type	1000 km

Carbon Dioxide Assessment and Optimization

Since version 2.2.0 the JOpt Vehicle Routing Optimizer comes with a built in emission assessment and optimization scheme that allows fleet operators to determine and optimize their overall carbon dioxide emission. Beside calculation of the overall and per route emissions, JOpt determines an optimum schedule with optimum energy efficiency in terms of truck's payload to empty ratio as well as overall road mileage. Hence vehicles are automatically dispatched in a way that leads to a minimum carbon dioxide emission while all other planning objectives like time windows, load capacities, working times and so forth keep being satisfied as well.

Calculation scheme

In order to discuss the various aspects of the emission assessment scheme we assume the following sample route:



As depicted above a truck starts with a load of 30t at the depot and returns empty. Alongside the route it drops and picks up some load, so that each route segment is performed with a different load. In the following section we will discuss different indicators that JOpt can automatically calculate on a route like above. These indicators may serve as a reasonable basis for further emission assessment and optimization.

Fuel Usage and Carbon Emission

These are the very basic values that can make up the starting point of any further assessment. In order to obtain these values JOpt requires some input about the vehicle's fuel type and average fuel consumption.

```
...
CapacityResource truck1 = new
CapacityResource("Truck1",50.1167,7.68333,12.0,1200.0,workingHours);

truck1.setCost(1000,1,1);
truck1.setFuelType(0);
truck1.setAvgSpecificFuelConsumption(32.0); // rough estimate
truck1.addCapacity(30);
double[] truck1InitialLoad = {30}; //
truck1.setInitialLoad(truck1InitialLoad)
...
```

If you desire more accuracy you can also setup a load specific fuel table for each truck, that defines the truck's average consumption with respect to load in different steps of load increments. The respective array may vary according to the level of detail required. JOpt will automatically calculate the respective load ratio intervals as follows:

$$Interval = \frac{100\%}{(length_{consumptionArray} - 1)}$$

For example adding the following lines of code:

```
...
double[] loadBasedFuelConsumption = {20.0,23.0,28.0,30.0,35.0};
truck1.setLoadBasedAvgSpecificFuelConsumption(loadBasedFuelConsumption);

truck1.addCapacity(30);
double[] truck1InitialLoad = {30};
truck1.setInitialLoad(truck1InitialLoad);
...
```

results in the following model:

load ratio	0%	25%	50%	75%	100%
Comsumption	20	23	28	30	35

If we desire a more detailed level we simply have to add additional values:

```
...
double[] loadBasedFuelConsumption = {20.0,23.0,28.0,30.0,35.0,37.0};
truck1.setLoadBasedAvgSpecificFuelConsumption(loadBasedFuelConsumption);

truck1.addCapacity(30);
double[] truck1InitialLoad = {30};
truck1.setInitialLoad(truck1InitialLoad);
...
```

Having added one additional value will result in a more detailed model:

load ratio	0%	20%	40%	60%	80%	100%
Consumption	20	23	28	30	35	37

In order for the detailed model to take effect we will also have to add load and unload information alongside the route:

```
...
TimeWindowGeoNode koeln = new TimeWindowGeoNode("Koeln",
50.9333,6.95,weeklyOpeningHours,1200,1);
double[] koelnLoad = {-5}; //
koeln.setLoad(koelnLoad);
this.addElement(koeln);
...
```

Load Distance

The load distance is given by the following quotation and gives an impression on how many physical work (force * distance) has been performed per route and by the overall fleet.

$$LDist = \sum_{i=0}^n dist_i * load_i$$

As above, load distance calculation requires a load and capacity model. This model can be entered as follows.

```
...
double[] loadBasedFuelConsumption = {20.0,23.0,28.0,30.0,35.0,37.0};
truck1.setLoadBasedAvgSpecificFuelConsumption(loadBasedFuelConsumption);

truck1.addCapacity(30);
double[] truck1InitialLoad = {30};
truck1.setInitialLoad(truck1InitialLoad);
...
```

```
...
TimeWindowGeoNode koeln = new TimeWindowGeoNode("Koeln",
50.9333,6.95,weeklyOpeningHours,1200,1);
double[] koelnLoad = {-5}; //
koeln.setLoad(koelnLoad);
this.addElement(koeln);
...
```

Having entered the capacity and load model, JOpt will optimize route distance and time while respecting capacity constraints. In addition JOpt calculates Load Distance and Energy Efficiency.

Energy Efficiency

Energy Efficiency is a value that easily figures out how effectively a fleet is operated. Having an energy efficiency close to 1.0 means that a fleet is mostly operated fully loaded whereas a value close to 0.0 means that the fleet is mostly driving empty or with unused shipping space which of course is not desirable in terms of efficiency.

Energy Efficiency is given by the following quotation:

$$\eta_{energy} = \frac{\sum_{i=0}^n dist_i * \frac{load_i}{capacity}}{dist_{total}} \leq 1.0$$

Where $dist_i$ and $load_i$ are each segment's load & distance and capacity is the truck's total load capacity.

Fuel & Emission per Load Kilometer

Fuel and Emission per Load Kilometer is another way to tell you how effectively your fleet is operated. The quotation is given herein:

$$\eta_{energy} = \frac{\sum_{i=0}^n dist_i * \frac{load_i}{capacity}}{dist_{total}} \leq 1.0$$

Obtaining the results

You can obtain the results on a overall and on route level. For overall results we simply add the following code:

```
Console.WriteLine ("\n#####");
Console.WriteLine("NEW OPTIMIZATION RESULT");
Console.WriteLine ("#####");

Console.WriteLine ("costfunction      : "
+ result.getTotalCost());

Console.WriteLine ("total route time   : "
+ result.getTimeTotal()/3600.0+" [h]");

Console.WriteLine ("total driving time : "
+ result.getTimeTrip()/3600.0 + " [h]");

Console.WriteLine ("total service time : "
+ result.getTimeStop()/3600.0 + " [h]");

Console.WriteLine ("total idle time    : "
+ result.getTimeIdle()/3600.0 + " [h]");

Console.WriteLine ("total break time   : "
+ result.getTimeBreak()/3600.0+ " [h]");

Console.WriteLine ("total distance : "
+ result.getTotalDistance()/1000 + " [km]");

Console.WriteLine ("total fuel use     : "
+ result.getTotalFuelUsed() + " [l]");

Console.WriteLine ("total CO2 emission : "
+ result.getTotalCarbonDioxideEmission()/1000 + " [kg]");

Console.WriteLine ("energy efficiency  : "
+ result.getEnergyEfficiency());

Console.WriteLine ("load distance : "
+ result.getTotalLoadDistance()/1000.0 + " [load*km]");

Console.WriteLine ("fuel efficiency : "
+ result.getFuelPerLoadKilometer() + " [l/(load*km)]");

Console.WriteLine("emission efficiency : "
+ result.getEmissionPerLoadKilometer() + " [g/(load*km)]");
```

For route level detail the API is quite similar. Please refer to the **CarbonEmissionExample** shipped with the SDK for more details.

What's next ?

All the above values are directly linked to the fleet's efficiency and build the foundation for further optimization that can be performed. Here are some ideas:

- increase drop density by adding additional freight alongside ineffective routes
- reorganize time windows,
- add additional resources
- remove resources
- try different truck types (e.g. different capacity, consumption)
- change resources costs
- change node priorities

Now you shall have understood the basic principle of emission assessment and optimization and should be ready to make your first steps. Of course whenever you feel that you need assistance or advice feel free to contact our technical support.