Route optimization for a fleet of vehicles with temporal constraints

---

Algorithmics Methods and Mathematicals Models

Prof. Albert Oliveras LLunell and Prof. Luís Velasco Esteban

Master in Research and Innovation

Juan Francisco Martínez Vera
`juan.francisco.martinez@est.fib.upc.edu`

January 20, 2017

Abstract

Optimization problems can appear in almost all situations on life. There are specially important those ones that appears on the industry because if we can achieve an optimal solution, we will improve the efficiency of the industrial process and then get lower manufacturing cost. This improvement of the costs have an effect on the competitiveness of the companies and on the final quality of their products and there are always good news for the customers. The challeging part is that those kind of problem have a really high computational complexity therefore there are several methods to face with them. By one hand we have the, let's say, always-optimality methods that obtain the optimal solution but without taking into account the huge amount of time that it could imply. e.g. ILP[1]. By the other hand we have those methods that concerns about the execution time and are looking for a tradeof between the exeution time and the quality of the solution[2].

In this pages will be shown this two approaches in order to get the optimal routes for a fleet of vehicles taking into account temporal constraints.

[1]Integer Linear Programming
[2]The distance to the optimal

CONTENTS

CHAPTER

1 INTRODUCTION

Optimization problems can appear in almost all situations on life. There are specially important those ones that appears on the industry because if we can achieve an optimal solution, we will improve the efficiency of the industrial process and then get lower manufacturing cost. This improvement of the costs have an effect on the competitiveness of the companies and on the final quality of their products. The challeging part is that those kind of problem have a really high computational complexity therefore there are several methods to face with them. By one hand we have the, let's say, always-optimality methods that obtain the optimal solution but without taking into account the huge amount of time that it could imply. e.g. ILP[1]. By the other hand we have those methods that concerns about the execution time and are looking for a tradeof between the exeution time and the quality of the solution[2].

In this project has been developed an Integer Linear Programming model in order to optimize the routes for a fleet of vehicles with temporal constraints derived from the distance between locations and from the tasks that has to be done for every location. Also have been developed different meta-heuristic strategies in order to get results more efficiently in terms of exeuction time even if there are not the optimal but close enough. Finnaly comparisons in term of execution time and solution quality have been done.

1.1 Document structure

The structure of this document is the following: The problem definition is done in chapter 2. Here it is explained the problem that is faced, which constraints is needed to take into account and what we want to optimize. At chapter 3 is explained the mathematical model that has been developed, i.e. the decission variables and also the constraints with a mathematical nomenclature. After this chapter, at chapter 4 is explained how it has been used the heuristics approach in order to face with the problem, two meta-heuristics has been used: GRASP[3] and BRKGA[4]. After perform several executions for those approaches, a comparison in terms of time and quality of the result is done at chapter 5. Finnally conclusions of the project are explained at chapter 6

[1]Integer Linear Programming
[2]The distance to the optimal
[3]Greedy Adaptative Search Procedure
[4]Biased Random-Key Genetic Algorithm

CHAPTER

2 PROBLEM DEFINITION

2.1 Description

We have been asked to help one logistic company in the design of the daily routes of its fleet of vehicles. The workload for every day consist in some tasks that have to be done at a different locations, therefore, the problem is to identify the optimal routes for the minimum number of vehicles in order to perform all the work in time, i.e. starting at 8 a.m. and finalizing before 8 p.m.

For that purpose we are given a set of locations and a start location l_s where all the vehicles will start. For our point of view, the locations are not needed at all but we need the distance in terms of time between all of them. Because that we will be provided by the $dist_{l_1,l_2}$ input data.

As is already mentioned, in every location (but not in l_s) there is a task that has to be done, then, also the information about how many time is needed for every task is provided as input data. We are talking about $task_l$.

Finally one more constraint is imposed by this logistic company. Every task has to be done but can not be started at any hour of the day. There is a temporal windows for every task that describes when an specific task can be started. This windows consists on a lower boundary min_l and on a upper boundary max_l. Then the starting time can not be before min_l and after max_l.

Considering that the company has an unlimited number of vehicles, the goal of the project is to find the minimum number of vehicles needed to visit all locations and perform all tasks, and define their routes. Given two solutions with the same number of vehicles it is preferred the one in which the latest vehicle arrives at its final destination sooner.

2.2 Formal definition

In this section is defined how the problem will be managed, also the input variables, the constraints and the objective function. This is a formal definition, therefore is implementation independant. These definitions have been used both for ILP and for meta-heuristics.

2.2.1 A relaxed TSP with temporal constraints

This problem has been formally defined as a relaxed TSP[1] with temporal constraints. It is relaxed because allows more than one cycle, i.e. more than one travel salesman (vehicle), and it is with temporal constraints because in addition to the usual weighted edges (routes) in the original TSP,

[1]Travel Salesman Problem

we have temporal windows and tasks times for every node (location). The key point is that all the cycles in the graph, i.e. all routes, **must imply the l_s**.

This point of view allows to manage this problem without taking into account vehicles. The number of vehicles will be, in fact, the number of cycles in the solution.

As in TSP we can represent the towns and all routes as a directed weighted complete graph, then, we can describe a route of every vehicle by mean of a set of edges that describes a cycle. Remember that the cycle must implie the l_s node.

The definition of the graph is defined by the below expression. Is important to mention that the cardinal of the A set is like that because the distance of the travel from location a to location b could be different than the travel from b to a because the roads or whatever else.

$$G = (V, A) \quad where \quad \text{V: set of nodes, A: set of directed edges}$$

$$|V| = n, \text{being n = number of locations} \quad |A| = n^2 - n$$

Now, let's define the set the whole possible cycles in the graph as C where every cycle is defined as a subset of edges that fulfill all the conditions to be a cycle. Then, the set of all possible vehicles $VH \subset C$ are all the cycles that imples the l_s as is described in the next expression:

$$vh \in VH \Leftrightarrow \exists a \in vh \text{ s.t. } a_{source} = l_s \vee a_{destination} = l_s$$

We can describe a set of potential solutions for the problem $PS \subset VH$ as all of subsets of VH that cover all the locations without repeat any one of them. Those are potential solutions because for the moment we have not token into account the temporal constraints. Finally, the set of feasible solutions will be $S \subseteq PS$. And the challenging point is to get the optimal solution from this set.

2.2.2 Input data

According the definition, this variables are provided and describes a single instance of this problem. The following list describe all the input data that is provided. In order to improve the understanding, the input data have been split into two subsets. By one hand all data related to the graph stuff, and by the other hand the input data related to temporal restrictions.

Graph input data

This input data is used in order to build up the graph that represents the problem and has been described several lines above. On next lines this input data is showed and explained.

- **Number of locations** that the vehicles have visit. From here we can define the V set that is the nodes set of the graph that are in fact the set of locations.

  $$n \in \mathbb{N}$$

- **Starting location** where all vehicles are placed at the beginning of times.

  $$l_s \in V$$

- **Distances** in minutes from one locations to the other. This data form the A set because it is the weights for the edges for our completed directed and weighted graph. Note that when $a = b$, then $dist_{l_a,l_b} = 0$

  $$dist_{l_a,l_b} \in A \quad \forall l_a, l_b \in V$$

  Could be said that this is a temporal data but as this data is also needed to build up the graph, and is present on the original TSP as well, is prefered to be here.

Time constraints input data

This input data is used in order to build up the time constraints needed for the problem definition. On next lines this input data is shoed and explained.

- **Lower boundary** for the temporal windows for every location. It indicates from when a task can be executed. If one vehicles arrives before this time, it must wait. The temporal unit are the minutes. This minimum windows can be at most 720 that is the journey time in minutes ($8h * 60min/h = 720min$)
$$min_l \in [0, 720] \quad \forall l \in V$$
- **Upper boundary** for the temporal windows for every location. It indicates until when a task can be executed. Take into account that does not have sense that $max_l < min_l$
$$max_l \in [min_l, 720] \quad \forall l \in V$$
- **Time** that a task spends for every location.
$$task_l \in [0, 720], \forall l \in V$$

2.2.3 Constraints definition

In this section are explained the fundamental constraints that are implementation independant. As well as with input data on the previous section, in order to improve the understanding, the constraints are splitted into two subsets, the graph constraint and the time constraints.

Graph constraints

These constraints are derived from the format description of the problem as a directed weighted graph. All the solutions that exists in set VH (section 2.2.1) fullfil the graph constraints. Just for summing up, those constraints are

- Every location has to be visited exactly one time. It means that all cycles in a solution must cover all nodes and can not share any node.
- All cycles in a solution must involve the starting location.

Time constraints

These constraints have to guarantee that the time restrictions are fullfiled. Those constraints are:

- The arrival time to location l must be less or equal max_l.
- The arrival time to location l must be $max(arrivalTime, min_l)$. It means that if one vehicle arrive before min_l it will wait.
- The total time of a cycle must be less than 720 minutes that is 8 hours.

Note that the arrival time to a location is the sumation of the previous location arrival time plus the task of the previous location plus the time spent in the path (the weight of the edge).

2.2.4 Objective function definition

In order to get the optimal solution we need to clasify every solution by a number that can tell us the quality of every one of them. This number is provided by the **objective function**.

As the description of the problem said, we want to minimize the number of vehicles and when two solutions have the same number of vehicles, we want the solution that end the whole job before. We will need two variables for this purpose, by one hand, the number of vehicles, lets say $nVehicles$ and by the other hand the time when the last vehicle ends its job, lets say $lastArrival$. How to get the correct value for these two variable is implementation dependant and therefore will be explained on the next chapters.

The objective functions is described as:

$$Minimize \quad nVehicles * M + lastArrival$$

In this expression, the M is a big enough number. It is needed in order to priorize the solutions with less value for $nVehicle$. When two solutions have the same number of vehicles, then the important value is the $lastArrival$.

CHAPTER

3 ILP MODEL

ILP[1] is the first of the two methods that has been used in this project. This method is always concerned on having the optimal solution without taking into account the amount of computational resources needed. This model is developed in CPLEX and the model implemented is described at the next sections.

3.1 Input data

From chapter 2 we already know which is the data that is provided. Now we need them in an specific shape in order to deal with them in CPLEX.

- **Number of locations** n is declared as an integer as well as **starting location**

$$n \in \mathbb{N}$$
$$l_s \in \mathbb{N} \text{ s.t. } 1 \leq l_s \leq n$$

- The **distances** are modeled as two-dimensional matrix of integers that have n^2 positions. If you take into account that the values on the diagonal of the matrix will not be taked into account, the total number of distances is $n^2 - n$ as we have described in the previous chapter. The distance is expressed in minutes.

$$dist_{a,b} \in \mathbb{N} \quad \forall a, b \in [0, n) \text{ s.t. } a \neq b$$

- The remaining data fields are related to the locations, because that they are defined as a vector on n positions. Those data fields are **lower boundary** and **upper boundary** of the temporal windows and the time spent in a **task**. All of them are expressed in minutes.

$$task_i \in \mathbb{N} \quad \forall i \in [0, n) \text{ s.t. } i \neq l_s$$
$$minw_i \in \mathbb{N} \quad \forall i \in [0, n) \text{ s.t. } i \neq l_s$$
$$maxw_i \in \mathbb{N} \quad \forall i \in [0, n) \text{ s.t. } i \neq l_s$$

[1]Integer Linear Programming

3.2 Decision variables

The solution of the problem will be encoded in a certain way in a decission variable. The decission variables are not only used for get the final result but also for enforcing some situation with constraints. Those variables unlike input data ones, are not statics and can change their value in order to fulfil the constraints (section 3.3), as is just said, and in the same time in order to maximize or minimize the objective function (section 2.2.4). Therefore, even if the important numbers for the solution are two. The number of used vehicles and the time when the last vehicle done the work is needed a sort of other decision variables. In the next list is presented a relation of the defined decision variable for model.

- The **tracked** variable. It is a nxn matrix and keep information about whether a path from location n to location $n2$ is taken, i.e. keep information about which edges are on the solution. It is needed in order to control the number of input and output edges for every location, i.e. to control the correctness of the cycles and for calculate the time spent in a route by determining which paths have been taken and which ones not.

$$tracked_{a,b} \in \mathbb{B} \quad \forall a, b \in [0, n) \text{ s.t. } a \neq b$$

- In order to build the temporal constraints that we have at section 2.2.3 we need information about when a vehicle arrives to a location. Because that the **arrivingTime** decision variable has been defined. It is an n-vector that keeps the arriving time for all locations (nodes) in minutes.

$$arriving_i \in \mathbb{N} \quad \forall i \in [0, n) \text{ s.t. } i \neq l_s$$

  The starting location will not have arriving time, i.e. it will be always 0. The reasons are explained at section 3.3.

- One of the most important decision variables is the numbers of vehicles needed. **nVehicles** keep the number of vehicles in the solution.

$$nVehicles \in \mathbb{N} \text{ s.t. } 1 \leq nVehicles \leq n$$

  On the best case, we only will need one vehicle but in the worst one we will need as much vehicles as locations.

- The other really important value for our result is the end time of the last vehicle. **lastDone** have the time in minutes of the last arrival vehicle.

$$lastDone \in \mathbb{N}$$

3.3 Constraints

For an ILP model the constraints part is the most important part, or at least the most tricky part. They are defining the solution space and because that we need to be careful in order to try to limit as much as possible but without prune some solutions and forbidding CPLEX to find a feasible solution when it exists.

The constraints below, as in section 2.2.3 have been separated into two sets. By one hand, the graph constraints and by the other hand, the time constraints.

3.3.1 Graph constraints

This constraints are the ILP version of the constraints defined at section 2.2.2.

- The first constraint is worried about the fact that every location has to be visited exactly one time. For this purpose the *tracked* decision variable is used. There is one special case where the number of visits for a location must be one or more. This special case will be the starting location because we want that all cycles involve it, therefore it will be visited at least one time.

$$\sum_{j \in V} tracked_{i,j} = 1 \quad \forall i \in V \text{ s.t. } i,j \neq l_s$$

$$\sum_{i \in V} tracked_{i,l_s} \geq 1 \quad \text{ s.t. } i \neq l_s$$

  The key point in this constraint is that we are counting the ingoing edges for all locations.

- From the previous constraint, we can ensure that every location just will have one ingoing edges but we are not ensuring anything about the outgoing edges. For cycles we want to ensure that there is only one outgoing edge for all location except for the starting one, that must be same outgoing edges as ingoing ones. Then, the sumation of outgoing edges must be equal as the ingoing edges for all locations.

$$\sum_{j \in V} tracked_{i,j} = \sum_{k \in V} tracked_{k,i} \quad \forall i \in V$$

- One desirable constraint is that we want to avoid the cycles that just imply one location. The point for this constraint is that we are enforcing 0 on the diagonal of the tracked matrix. As we will see later, since we are enforcing that all cycles must imply the startLocation, this constraint is not needed at all.

$$\sum_{i \in V} tracked_{i,i} = 0$$

The constraints above are ensuring that our solution will contain cycles, and those cycles will visit all locations. Also is ensuring that all cycles are independant betweem them. The important thing that those constraints are not ensuring is that all cycles implies the starting location. This requirement will be fulfilled by time constraints as you will see on the next section.

3.3.2 Time constraints

This is the ILP implementation for the constraints defined at section 2.2.3. As you will see below, this constraints are not enough, so some other have to be defined.

- As we have seen at section 3.2 just with the obvious decision variables is not enough in order to maintain all the constraints of times fulfilled. Because that `arrivingTime` decision variable is needed. Since we have this variable is needed to constraint it, otherwise it will take whatever value. At every position of this vector we want to have the arriving time for the location with a determined id. If we think a little bit, there is one location that we already know its arriving time, this is the start location, and its arriving time should be zero and this constraint is enforcing this. This constraint is more important that it seems in principle. The implications are explained at section 3.3.3.

$$arriving_{l_s} = 0$$

- Continuing with the arriving time, we also want good values for the rest of locations. In this constraint the arriving time for every locations is computed as the arriving time of the previous location plus the task done in that location plus the distance in minutes of the path that connects this pair of locations. Then, we must use the distances input data and also the decision variable `tracked` in order to know which is previous location of the one that we are restricting.

$$\sum_{i \in V} arriving_j \geq (arriving_i + task_i + distance_{i,j}) - (M * (1 - tracked_{i,j})) \quad \forall j \in A \text{ s.t. } j \neq l_s$$

For this equation, i means the source location and j means the destination. Take into account that we are avoiding the constraints for the l_s because we are already restricting it to 0 with the constraint above. The key point is that when the path from i to j is done (tracked is True) then we have a strong constraint[2]. By the other hand, if the path is not token then this is so soft constraint, in fact this is not restricting anything.

- This next constraint is concerned about the minimum windows arriving time. The point is that if the vehicle arrives before the minimum time, then it has to wait. The way to do so is restricting that the arriving time will be at least the minimum window. It is in fact a waiting.

$$\sum_{i \in V \text{ s.t. } i \neq l_s} arriving_i \geq minw_i$$

- And what about *maxw*? This constraints avoid those solutions that does not fulfil the window time requirements.

$$\sum_{i \in V \text{ s.t. } i \neq l_s} arriving_i \leq maxw_i$$

- Until here the model has been concerned about having good values for this, let's say, internal decision variable but what about the really important ones?. In this constraint we are ensuring a good value for the $nVehicles$ variable. In this model the number of vehicles are the number of cycles, so we just have to count how many cycles we have. To do so, an easy way is just count how many ingoing or outgoing edges we have for the start locations.

$$nVehicles = \sum_{i \in V \text{ s.t. } i \neq l_s} tracked_{l_s,i}$$

- The last part of this model is to ensure that the whole work is done before the end of the working day. To do so we need to have a good value for the *lastDone* variable. In this model *lastDone* is calculated by getting the arriving time for all of the locations that have an outgoing edge to start location, for every one of them do the summation of the task and the distance to start location and then get the biggest value.

$$\sum_{i \in V} lastDone \geq (arriving_i + task_i + distance_{i,l_s}) - (M * (1 - tracked_{i,l_s}))$$

- To end with this section, the last constraint is forbidding the working days over the eight hours (720 minutes).

$$lastDone \leq 12 * 60$$

3.3.3 Avoiding cycles w/o starting location for free

At section 2.2.1 has been explained how a vehicle is in fact a cycle. The important point is that all cycles represents all vehicles needed in our model and they must imply the start location. As you can see at section 3.3 there is no explicit constraint that enforce this situation, so how it is actually working?

Imagine a situation in which one we have two locations in a cycle and no one of them are the starting. And now take into account the way this model is enforcing a good value for the arriving

[2]The constraint is great or equal because the arriving time could be greater than that if the vehicle must wait because the *minw*.

time for locations. Is not possible to have an arriving time in any of them because there is not a fixed value, so the arriving time for both of them will be undefined, in other words, is not a solution.

By enforcing the arriving time to 0 at starting location we are providing an exit point where all locations can go. Therefore, no cycle w/o starting location will form part of a valid solution, i.e. All the cycles of a valid solution will imply the starting location.

CHAPTER

4 META-HEURISTICS

In this chapter are explained via pseudocode the two meta-heuristics used for solve the problem. Only the most important parts will be shown, ignoring the rest of the code.

For GRASP the constructive phase and local search algorithms will be explained and for BRKGA, the chromosome structure and the decoder.

Just mention that the shape of the results for the meta-heuristics are the same as for ILP, that is the shape that has been described at section 2.2.1. It is a set of cycles, i.e. a set of paths

4.1 GRASP implementation

4.1.1 Costructive phase

This meta-heuristic has a constructive phase that is concerned to build up a feasible solution. This phase is not deterministic because the randomization done on the candidate pick process. It means that for every execution, different solution could be emerge[1]. This part is based on a randomized greedy algorithm, so a greedy function is also needed.

You can see the constructive phase into two pseudo-codes. The constructive phase itself is at psheudcode 4.1.1. But in order to understand better what the developed heuristic is doing, you can see how the candidates set is generated at pseudocode 4.1.2.

The essential idea is that a vehicle is moving through locations one after the other until it can not go to the next one because the time restriction. In this case, it returns to the l_s. Notice that at candidate generation is checked if moving to this next location there is a possibility to return to l_s, if not it is not considered a candidate.

4.1.2 Local search

The other part that will be shown is about the local search. Once the constructive phase ends up with a solution, eventually it is the optimal, but usually it is not. In order to improve the solution a neighbourhoods of the constructive phase solution, i.e. near solutions will be searched. This phase is called local search. Two neighbourhoods have been developed. The exchange and the reassignement. You can see the first one at pseudocode 4.1.3 and the second one pseudocode 4.1.4.

[1]For $\alpha \neq 1$

Exchange neighbourhood

This neighborhood is formed by all near solutions from the solution done by the constructive phase that differenciates with it because there is one location for one vehicle that has been changed for other location of other vehicle. Notice that searching in this neighborhood will not bring us a better solution in terms of number of vehicles but just in terms of final arrival time.

Reassignement neighbourhood

In this case, the differences between the solution at constructive phase and the solution on this neighbourhood is that one of the locations of one vehicle is not visited by this one but by other one. Notice that in this neighborhood is possible that we can have both, a better solution in terms of number of vehicles and in terms of final arrival time.

4.1.3 Greedy function

Finally, an important function for this meta-heuristic is the greedy function. This greedy function is giving us information about the quality of a candidate in order to decide if this candidate should be part of the solution or not. In this case, the greedy value depends on the characteristics of the path[2], the characteristics of the locations that this path connects and also on the progression of the travel, i.e. on the travel time.

$$q(c, TravelTime) = TravelTime - distance(c) - minw(destination(c)) + task(source(c))$$

This greedy function is trying to give more importance to those candidates that implies less waiting time, i.e. it is trying to maximize the efficiency of the solution.

Algorithm 4.1.1: GRASP CONSTRUCTIVE PHASE($\alpha, Paths, Locations$)

```
$S \leftarrow \varnothing$
$LastLocation \leftarrow l_s$
$VisitedLocations \leftarrow \varnothing$

while not $size(VisitedLocations) = n - 1$
  do
    $TravelTime \leftarrow tavelTime(S)$
    comment: The candidates are not locations but paths
    $C \leftarrow ConstructCL(LastLocation, TravelTime, VisitedLocations)$
    if $size(C) = 0$
      then return $(infeasible)$

    comment: Notice that q(c) returns the greedy value for candidate c.
    $q_{min} \leftarrow min(q(c, TravelTime) \quad \forall c \in C)$
    $q_{max} \leftarrow max(q(c, TravelTime) \quad \forall c \in C)$
    $RCL \leftarrow c \in C$ s.t. $q(c, TravelTime) \leq q_{min} + \alpha * (q_{max} - q_{min})$
    select $c \in RCL$ at random

    $S = S \cup c$
    if $destination(c) \neq l_s$
      then $VisitedLocations = VisitedLocations \cup destination(c)$
    $LastLocation \leftarrow destination(c)$

comment: The return path to the start location for the last vehicle must be added.
$ReturnPath \leftarrow getPath(destination(c), l_s)$
$S = S \cup ReturnPath$
return $(S)$
```

[2]The candidates are not locations but paths

Algorithm 4.1.2: CONSTRUCTCL($FromLocation, TravelTime, VisitedLocations$)

```
$C \leftarrow \varnothing$
$paths \leftarrow pathsFrom(FromLocation)$
comment: Filter unattainable locations
for each $c \in paths$
  do if $maxw(destination(c)) \geq TravelTime + task(FromLocation) + distance(c)$
  then $C = C \cup c$
comment: Filter visited locations
for each $c \in C$
  do if $destination(c) \in VisitedLocations$
  then $C = C - c$
comment: Filter those locations that if go there is not possible to return to $l_s$
for each $c \in C$
  do
    $path_1 \leftarrow getPath(FromLocation, destination(c))$
    $path_2 \leftarrow getPath(destination(c), l_s)$
    $time_{path_1} = max(TravelTime, minw(FromLocation)) + task(FromLocation) + distance(path_1)$
    $time_{path_2} = max(time_{path_1}, minw(destination(c))) + task(destination(c)) + distance(path_2)$
    if $time_{path_2} > 720$
      then $C = C - c$
return (sorted $C$ by q(c))
```

Algorithm 4.1.3: EXPLOREEXCHANGENEIGHBOURHOOD($S, strategy$)

```
comment: The solution is a set of edges. This set of edges can be organized as cycles (vehicles)
$V \leftarrow vehicles(S)$
sort $V$ by $ArrivalTime$ desc

comment: Get the most loaded vehicle
$MLVehicle = V[0]$
$MLVehicleLocations \leftarrow locations(MLVehicle)$
sort $MLVehicleLocations$ by $task$ desc

comment: Now get the locations of the rest of vehicles sorted by task asc
$restOfLocations \leftarrow \varnothing$
for each $v \in vehicles(S)$
  do if $v \neq MLVehicle$
  then $restOfLocations \leftarrow locations(v)$
sort $restOfLocations$ by $task$ desc

$bestSolution \leftarrow S$
for each $highLoadedLoc \in MLVehicleLocations$
  do
    for each $lowLoadedLoc \in restOfLocations$
      do
        $neighboor \leftarrow generateExchangeS(highLoadedLoc, lowLoadedLoc)$
        if $feasible(neighboor)$ and $quality(neighboor) > quality(bestSolution)$
          then
            if $strategy$ = "FIRST-IMPROVEMENT"
              then return ($neighboor$)
              else $bestSolution \leftarrow neighboor$
return ($bestSolution$)
```

Algorithm 4.1.4: EXPLOREREASSIGNEMENTNEIGHBOURHOOD($S, strategy$)

```
comment: The solution is a set of edges. This set of edges can be organized as cycles (vehicles)
$V \leftarrow vehicles(S)$
$vehiclesByLoad \leftarrow$ sort $V$ by $ArrivalTime$ asc
$vehiclesByLocation \leftarrow$ sort $V$ by $nLocations$ asc

$bestSolution \leftarrow S$
for each $lessnLocactionsVehicle \in vehiclesByLocation$
  do
    for each $path \in lessnLocationsVehicle$
      do
        $locationToReassign = source(path)$
        if $locationToReassign \neq l_s$
          then
            for each $lessLoadedVehicle \in vehiclesByLoad$
              do if $lessLoadedVehicle \neq lessnLocationsVehicle$
                then
                  comment: Try to inject the location to some place of vehicle route
                  for each $pathWhereInject \in lessLoadedVehicle$
                    do
                      comment: If path is A-B and location is C, then try to A-C-B
                      $neighbor \leftarrow generateReassignementS(locationToReassign,$
                                  $pathWhereInject)$
                      if $feasible(neighboor)$ and
                                  $quality(neighboor) > quality(bestSolution)$
                        then
                          if $strategy =$ "FIRST-IMPROVEMENT"
                            then return $(neighboor)$
                            else $bestSolution \leftarrow neighboor$
```

4.2 BRKGA implementation

4.2.1 Chromosome structure

For this implementation, the chromosome will encode the order in which the locations are pushing to the solution, in fact is a factor for the greedy value. The decoder is behaving as a greedy algorithm but with some modification in order to work with the chromosome as you will see at section 4.2.2.

Then the chromosome will have as many gens as locations, and every gen will have a value from 0 to 1. This value is the weight that modify the order of the candidate set at decoder.

4.2.2 Decoder

Recapping, the greedy algorithm is based on the idea to bring for every candidate a value that means the quality of every one of them. This value is provided by the greedy function. For purely greedy algorithms, the candidates are sorted by this value and then the best one (the first one of the list) will be included on the solution. An evolution of greedy is GRASP that is randomizing the pick of a candidate with its RCL. This decoder is following the same idea. In this case, every gen is multiplied for the greedy value of the destination of every candidate (remember that the candidates are paths). It means that the gens are affecting to the position of the candidates in the candidates list. Once we have all the candidates sorted, the first one is picked. Notice that the decoder is deterministic. The randomicity in BRKGA is not in the greedy part but in the generation of mutants.

You can see the pseudocode of decoder at 4.2.1

Algorithm 4.2.1: ExploreReassignementNeighbourhood(*chromosome*)

```
$S \leftarrow \varnothing$
$LastLocation \leftarrow l_s$
$VisitedLocations \leftarrow \varnothing$

while not $size(VisitedLocations) = n - 1$
  do
    $TravelTime \leftarrow tavelTime(S)$
    comment: The candidates are not locations but paths
    $C \leftarrow ConstructCL(LastLocation, TravelTime, VisitedLocations)$
    if $size(C) = 0$
      then return $(infeasible)$

    comment: getGen(chromosome, location) provides the value of the gen for location.
    $c \leftarrow min(q(c, TravelTime) * getGen(chromsome, destination(c)) \quad \forall c \in C)$

    $S = S \cup c$
    if $destination(c) \neq l_s$
      then $VisitedLocations = VisitedLocations \cup destination(c)$
    $LastLocation \leftarrow destination(c)$

comment: The return path to the start location for the last vehicle must be added.
$ReturnPath \leftarrow getPath(destination(c), l_s)$
$S = S \cup ReturnPath$
return $(S)$
```

CHAPTER

5 TUNNING AND COMPARISONS

There are some interesting comparisons that can be do, not just compare ILP with meta-heuristics solutions. An interesting studies are for try to tune accuratelly the meta-heuristics parameteres. In this chapter you will see how those parameteres are tuned through an empirical reasoning. An study of the α value for GRASP is done at section 5.2. More parameters have to be taken into account for BRKGA, that are: percentage of elites, percentage of mutants, number of individuals, percentage on inheritage and number of generations. In this case some simplications have done in order to fulfil with the deadline. You can see this study at section 5.3. Finally, at section 5.4 a comparison between ILP and meta-heuristics is done.

5.1 Instances complexity factors

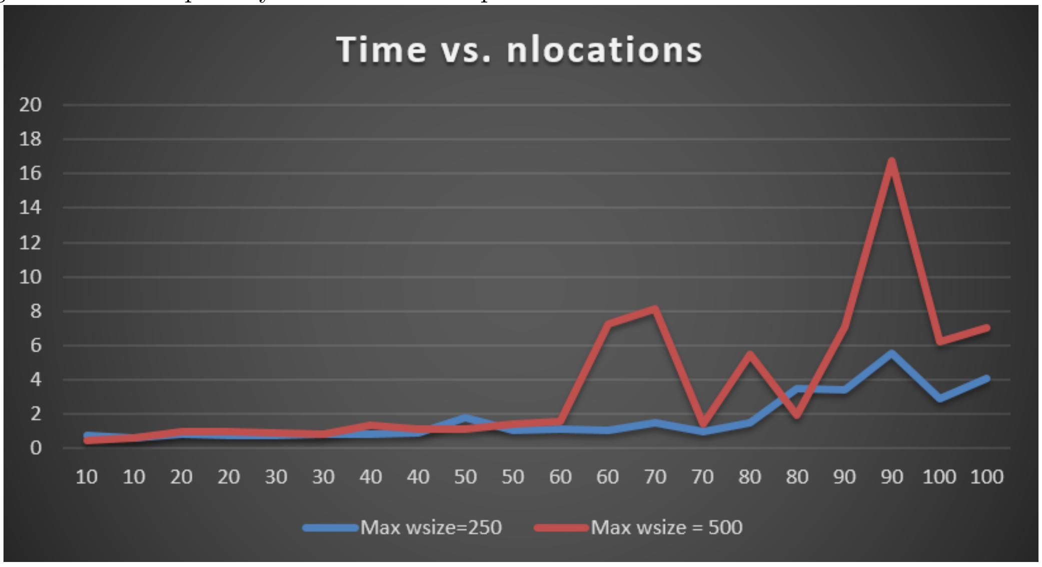
Fisrt of all a little study has been done in order to know in a better way which factors made a problem, computational power hungry or not for ILP. A naive approach could be to say just the number of locations. But there are more factors that affects on the complexity of the problem. Since ILP is taking into account all the possible solutions and then get the best one, when you are providing to the ILP model an instance where almost all combinations can be done, it will take a lot of time. So, three principal factors have to be token into account.

- Number of locations. Of course, as more locations we have, more combinations of routes can be done.
- The size of the time window. The time window is a good tool in order to prune options for routes. As more big is the windows of the locations more posibilities to move from one location to another the model has. As more little one, less posibilities to move.
- The distances between the cities. Also an obvious factor. As more scattered are the locations on the a bidimentional space, less jumps from one location to another can be done.

At plot 5.1 it can be seen the results for some experiments that are done in order to see the impact of the windows size. All these experiments are done with scattered distribution of locations. The conclusion is that when we have a little amount of cities, the windows size is not so important, but for bug amount of cities it affects a lot.

Has been decided to do this little study, that just take some minutes thinking on the problem, in order to end up with scaled instances for perform the experiments. We want complex enough inputs but not so complex that could imply more time that we have. To do so, some modifications in the `InstanceGenerator` have done. Now you can decide the number of locations, of course, but

Figure 5.1: Time spent by ILP for different problem size for windows time at max 350 and 500



also the maximum window and if the cities will be scattered on a big space or in a little one. The big space implies that the starting locations will be at the middle and the other will be as maximim at 360 minutes from it. The small space implies that the start location is placed randomly and then the maximum distance between all locatios is 360 (just the minutes that we need to go and return assuming task of 0 minutes).

In conclusion a tradeoff between number of locations, windows size and how to distribute the locations must be done. Since this point all the instances used for the executions have been generated taked into account those factors.

5.2 Tuning α parameter for GRASP

On GRASP, a good tune of α parameter is a critical issue. When this parameter is 1 the metheuristics becomes a completelly random algorithm. By the other hand when it becomes 0, the algorithms will behave as a purelly greedy one.

Some tests has be done with different number of locations for different α. 20 tests are done. 4 test por every number of locations. And the number of locations as gone from 10 up to 100.

As you can see in figure 5.2 the most suitable α for this problem is near $0,25$ then we will use this value for the next experiments.

This plot confronts the quality of the solution and the α parameter. For this experiment has been toked the best solution of the four experiment for each α and number of locations and compare the rest of the solutions whith same parameters which this one.

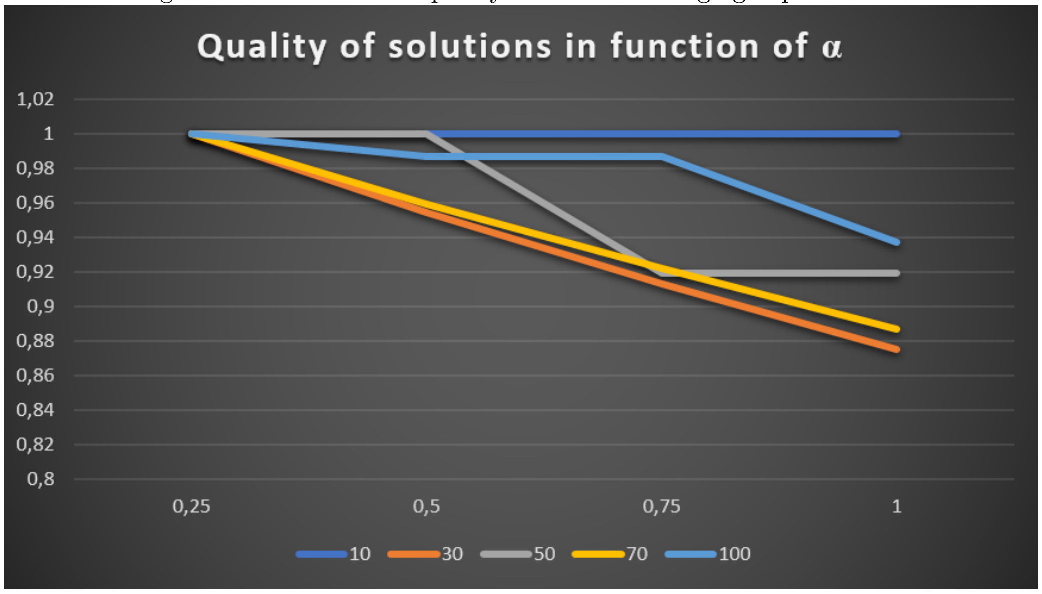
One interesting fact that also can be seen on the plot is that as we are incrementing the number of locations, the degradation because α is also increasing. An explanation for this fact is the synthetic instance generator. The instance generator developed for this project is just ensuring that you will have at least the basic solution that is have as much vehicles as locations. After ensure that it relaxes some data by randomizing values like task and time windows. When you have a little ammmount of locations this relaxation is not enough and the number of possible solutions is still near 1. Since the number of locations is incrementing, then the number of possible solutions is also incrementing and by get bigger α we are seeing those other solutions.

5.3 Tuning BRKGA parameters

For BRKGA the tuning is a little bit more difficult than for GRASP. Here we have five parameters to tune.

- *nIndividuals* Number of individuals in a poblation.

Figure 5.2: Evolution of quality of solution changing α parameter.



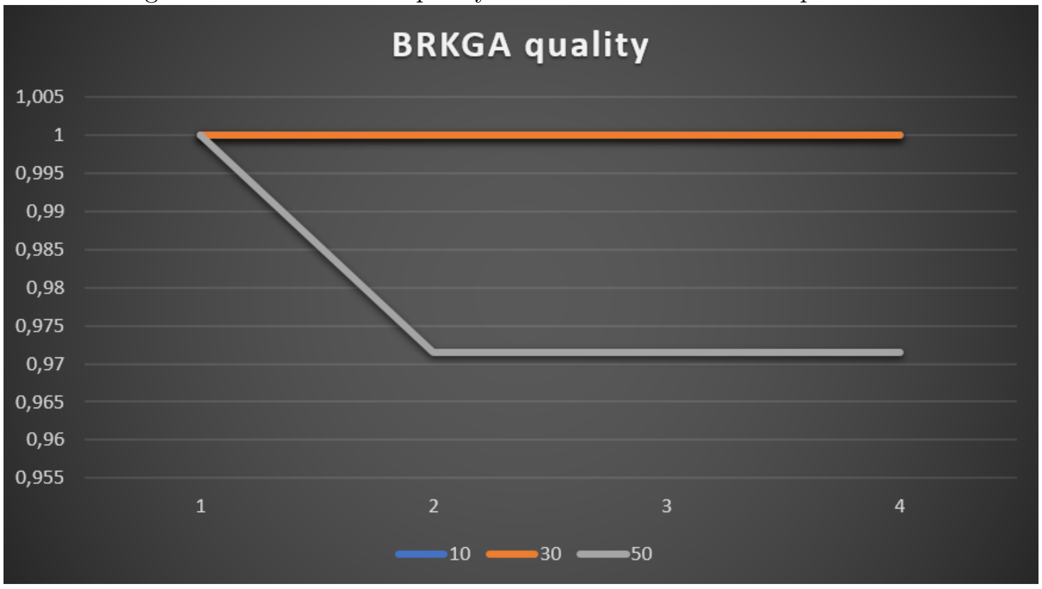
- *pElites* Percentage of the poblation that becomes elite.
- *pMutants* Percentage of poblation to generate as mutants.
- *pInheritanceElite* Boundary value that decides if a gene is get from elite or from a normal individual on the crossover.
- *maxGenerations* And finally the number of generation that we want to evolve.

In order to fulfil with the deadline of the project all the needed experiments are not done but just a little subset of them. It has been defined three different sets of parameters and every set has been executed over the same instances like the experiments for the GRASP at section 5.2 from 10 to 50 locations.

Those four sets are:

- Let's start with the configuration provided by default with the meta-heuristics framework. This configuration is:
  - $nIndividuals = nLocations/2$
  - $pElites = 0.2$
  - $pMutants = 0.15$
  - $pInheritanceElite = 0.7$
  - $maxGenerations = 3$
- Now let's try incrementing the number of individuals. This experiment is trying the exploit the generation of mutants in order to leave of local optimals.
  - $nIndividuals = nLocations$
  - $pElites = 0.2$
  - $pMutants = 0.15$
  - $pInheritanceElite = 0.7$
  - $maxGenerations = 3$
- In this experiment is almost equal the previous one but doubling the number of elite individuals.
  - $nIndividuals = nLocations$

Figure 5.3: Evolution of quality of solution for different experiments.



  - $pElites = 0.4$
  - $pMutants = 0.15$
  - $pInheritanceElite = 0.7$
  - $maxGenerations = 3$

- In this experiment is almost equal the previous one but doubling the number of elite mutants.
  - $nIndividuals = nLocations$
  - $pElites = 0.2$
  - $pMutants = 0.3$
  - $pInheritanceElite = 0.7$
  - $maxGenerations = 3$

As you can see at figure 5.3 the quality of the solutions are not changing at all. Only for the last input of 50 locations. Then we will use the first experiment parameters. The fact that there are not so many changes with the different parameters also could imply that the inputs are really constrained and encourage us to keep developing the `InstanceGenerator` for try to relax a little bit more the inputs.

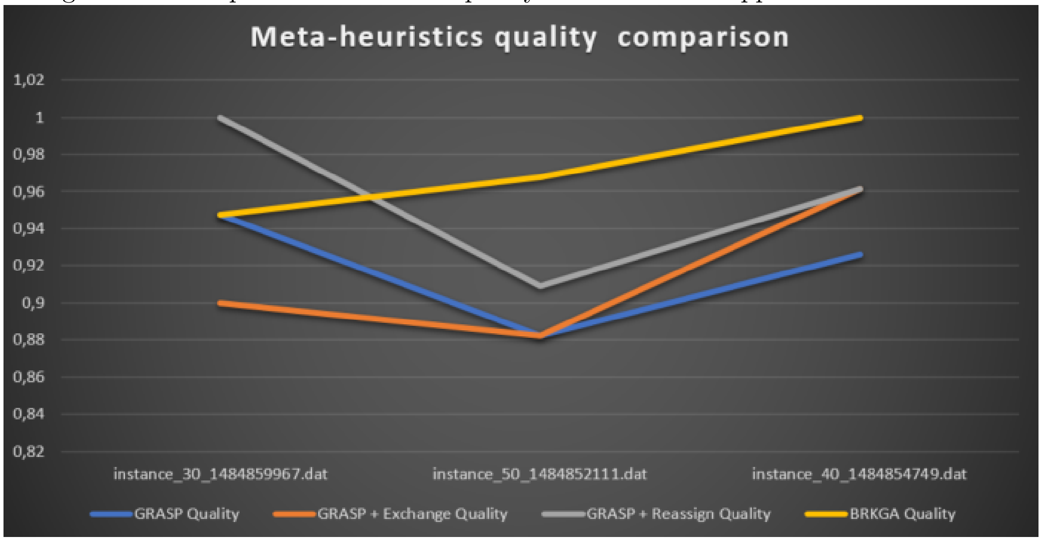
5.4 Comparison: ILP vs meta-heuristics

Now, a comparison in terms of time and quality of the result will be showed up. The quality of a result is the distance to the optimum. Taking into account that the solution of ILP is always the optimum, just have been calculated the quality for the two metaheuristics.

The tricky part in order to do this comparison has been pick the correct instances. As it has been explained at section 5.1 get the correct instances is not just a matter of move over the different intances sizes in terms of number of locations. Because the randomness of the instance generation is difficult to get the perfect input for the experiment that you want to do even if you are controlling all the explained factors.

The used instances have been instances that have an increasing time in ILP. This is just to see what is the evolution of execution times of meta-heuristics and also the evolution of the quality of the solutions since the end of this section is to make comparisons, not denote the evolution of times vs. input sizes. Is because that you will see that the instances on the x axys are unsorted in terms of number of locations.

Figure 5.4: Comparison in terms of quality of the different applied meta-heuristics.



5.4.1 Quality comparison

At figure 5.4 you can see the plot with the qualities of the solutions for four different instances. As is just said, the instances are not sorted by the number of locations but by the time spent in ILP because is considered that this time is a clear indicative for the complexity of the instance of the problem. Looking at the quality, it seems that as more number of locations we have, more further from the optimality the meta-heuristics are and this is the expected phenomena. Also another appreciation arise. Those values are saying us, at least for the different version for GRASP, that the complexity of an instance from the point of view of it is clearly correlated with the number of locations and it is saying that in fact its possible that the instances complexity factors described at section 5.1 are not aplying completely to meta-heuristics.

So, the conclusions for this comparison is that in general BRKGA is behaving better than GRASP. And for the different versions of GRASP, the best is GRASP+Reassignement. This is also as we are expecting because as has been explained at secion 4.1.2 scan on reassignement neighbourhood can end up a better solution by removing one vehicle. For the exchange neighbourhood this kind of improvement is not expected but just an improvement in terms of a better arrival time.

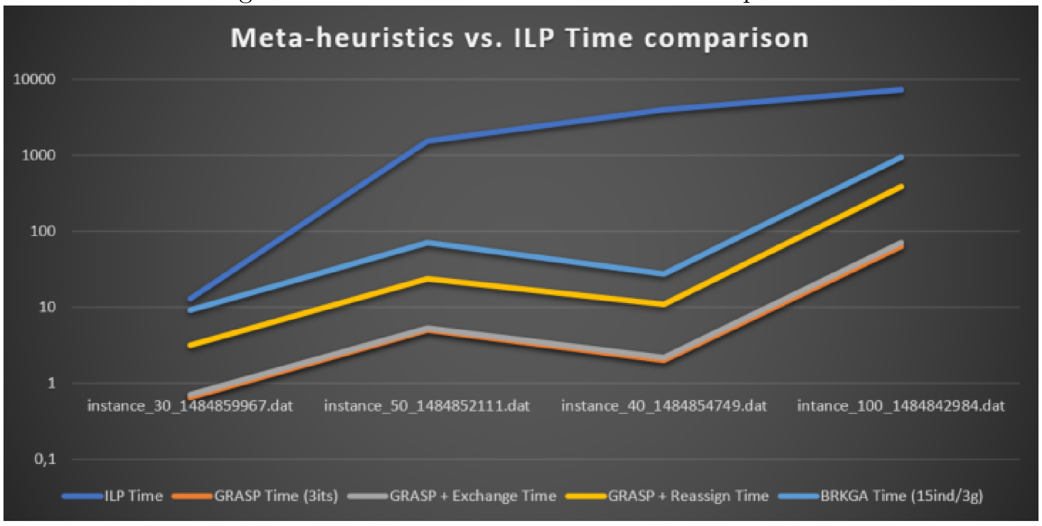
5.4.2 Time comparison

At figure 5.5 you can see the plot with the comparisons of execution times. Be careful for a fast conclusion just looking for lines, take into account that the y-axys is in a logarithmic scale. You can see, as was expected that ILP is the clear defeated because looking at figure 5.4, those times are not justified. The quality of the solutions are not so distant, so the efficiency (let's say $\frac{quality}{neededresources}$) of ILP, at least in these experiments is clearly below the efficiency of meta-heuristics.

About the meta-heuristics there are some things to mention. All the versions are paying the better quality with execution time. In fact the order is inverse than the order of the best meta-heuristic for quality. The differencies between GRASP+Exchange and GRASP+Reassign can be explained because the size of the neighbourhood that every one is scanning. Of course, the reassignement one is more bigger than the other.

Finnaly, the same appreciation like on the section above. It seems that the number of locations are strong correlated with the needed resources for meta-heuristics but not for ILP.

Figure 5.5: ILP vs Meta-heuristics times comparison



CHAPTER

6 CONCLUSIONS

In this project has been developed an ILP model and two meta-heuriscs in order to find out the optimal paths for a fleet of vehicles taking into account time constraint.

About the ILP has been really challenging to end up with a really simplified model that does not know anything about vehicles. This simplicity allows to model to work with two-dimensional matrix instead of three-dimensional ones that will imply more time resources for get the optimal value. Also an extra effort has been done for give it flexibility in terms on confire where the starting location is. And keeping on with the simplicity idea, this model enforce the generation of cycles without an explicit constraint. This is because some time constraint as has been explained. It was for free because the idea was to work on this constraint later.

About meta-heuristics, has been also so challeging to end up with a reasonable greedy function, that is in fact the hearth of all applied metaheuriscis here.

But the more challenging part has been to get a good inputs for the experiments. Has been so difficult because the complexity is not only a matter of number of locations but also another factors. The difficulty was because it can be generated two different instances with the same number of locations and one of them finalize at seconds and the other take ours until the computer run out of memory. Finally some good inputs has been picked by tunning the `InstanceGenerator`.

About the results they are so satisfactory because they are approximatelly as I was expecting. Anyway has been a surprise for me the power of the meta-heuristics. They can achieve a results really close to the optimal in a few minutes or even seconds while ILP is taking even hours. One commentary here for the executions of ILP. There are some situacions where it conversge very rapid to a less than 10% but it is running for hours just for this 10%. Maybe a more fine tunning of cplex in order to configure the minimum quality of results will save a lot of computational time.

6.1 Future work

More studies in order to get the optimal parameters for the meta-heuristics have to be done. Also one limitation of the experiments done in this project is that all the instances are generated with synthetic data, it means that of course they are biased by the nature of the generator. So it could be interesting to work with real data.

APPENDIX

A INSTANCE GENERATOR

For testing the ILP model and the meta-heuristics and perform the experiments an instance generator has been done. This instance generator allows us to generate feasible inputs. You will see the input generator script in attached with this report.

The usage of the script is:

```
python ./InstaceGenerator.py <nlocations> <maxwinsize> <SCAT | GATH>
```

The different parameters are:

- `nlocations`: For sure, the number of locations that you want on the instance.
- `maxwinsize`: The maximum sizes of the time windows. It means that the size of the windows of all locations will be set randomly from 1 to this number.
- `<SCAT | GATH>`: You must choice between scattered distribution or gathered. For scattered the nlocations-1 locations will be dispersed over a bidimensional are where the distance from the center to one corner is exactly 720/2. Since the startLocations will be on the center of the area, there is ensuring that all distributions will end up with a feasible solution because always a vehicle can travel from startLocation to the most remote location. For the gathered distribution the distance of the diagonal is 720/2 and all locations are distributed randomly. It means that the maximum distance between two locations will be 360.

Finally, the script will generate an instance with the name:

```
instance_<nlocations>_<random_seed>.dat
```

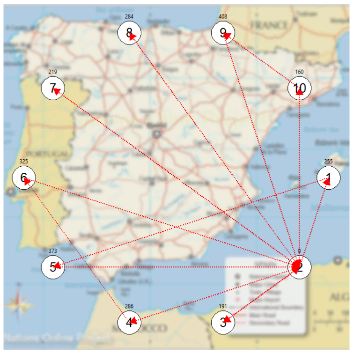
APPENDIX

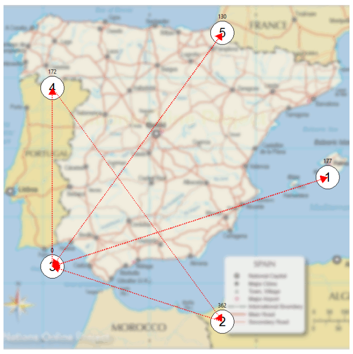
B RESULT VISUALIZER

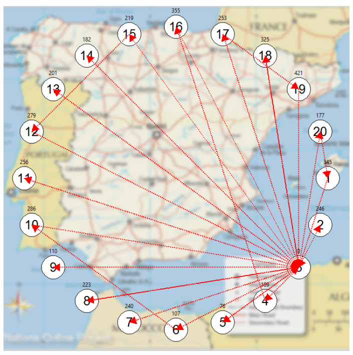
As an extra work, a basic visualizer has been developed. It shows the results by reading the output of ILP (previously saved on a file). The representation of the result is with directed graph and also can be seen the arrival time for every locations. To see the graphical representation of a solution you must execute:

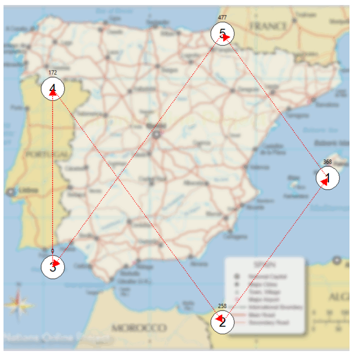
```
python ./ResultVisualizer.py <file_with_dumped_result.txt>
```

There are some examples below:


(a) Example 1


(b) Example 2


(c) Example 3


(d) Example 4