Alma Mater Studiorum - Università di Bologna

# **Project Work**
# Combinatorial Decision Making and Optimization

Multiple Couriers Planning

**Group:**
Davide Buldrini (davide.buldrini@studio.unibo.it - 0001026874)
Nicolò Donati (nicolo.donati4@studio.unibo.it - 0001045676)
Guido Laudenzi (guido.laudenzi@studio.unibo.it - 0001033343)

September, 2022

Master of Artificial Intelligence

Academic year 2021/2022

## Contents

# 1 Introduction

## 1.1 Problem description

The **Multiple Couriers Planning** (MCP) is a problem in which $m$ couriers must distribute $n \geq m$ items at different customer locations. Each courier has a fixed maximum load capacity. Each item has a distribution point and a size, which can represent for instance a weight or a volume.
The goal of MCP problem is to decide, for each courier, the items that should distribute and plan a tour (i.e. a sequence of location points to visit). Each courier tour must start and end at a given origin point (a depot). Moreover, the maximum load of the courier must be respected when items are assigned to it. The objective is to minimize the total tour distance.

MCP is a different name for the so-called **Vehicle Routing Problem** (VRP) which is a very common and known combinatorial optimization and integer programming problem. It generalises the **Travelling Salesman Problem** (TSP).
Determining the optimal solution to a VRP is NP-hard, so the size of problems that can be optimally solved using mathematical programming or combinatorial optimization may be limited. Therefore, commercial solvers tend to use heuristics due to the size and frequency of real world VRPs they need to solve.
VRP has many direct applications in industry: vendors of VRP routing tools often claim that they can offer cost savings.

There are a variety of VRP problems that have different implications. In this case, couriers' capacity can be different, so it is a **Multiple Capacity Vehicle Routing Problem** (MCVRP). A famous alternative is the one involving time windows (VRPTW).

## 1.2 Input format

Given $m$ couriers and $n$ items, an instance of MCP is a text file with the form:

$m$
$n$
$l_1\ l_2\ \dots\ l_m$
$s_1\ s_2\ \dots\ s_n$
$dx_1\ dx_2\ \dots\ dx_n\ ox$
$dy_1\ dy_2\ \dots\ dy_n\ oy$

Where $l_i$, $s_j$ are respectively the load of the courier $i$ and the weight of the item $j$; $(dx_i, dy_i)$ are the coordinates of the distribution points and $(ox, oy)$ are the coordinates of the origin point.
The Manhattan distance has been used to create the distance matrix $D$ of order $n+1$ where $D_{i,j}$ is the distance between point $i$ and point $j$. Depending on the paradigm considered, the origin could be considered either as the node $n+1$ or the node 0.

### 1.3 Pre-processing

Pre-processing involves all the modifications used to convert the blank instances to instances ready to be fed to the different models. For instance, the distance matrix is created at this stage starting from the given coordinates.

A useful pre-processing technique which was able to increase the performance of all the models on big instances, is the one of the **triangle inequality**: it aims to reduce the number of couriers, keeping only the ones that can handle all the packages. In fact, triangle inequality tells us that the more the couriers are used, the higher will be the distance traveled.

During pre-processing, it is also possible to find a satisfiable tour that could be used as a starting solution for the search thanks to the Warm or MIP start. For further information, see Warm Start paragraph in MIP model.

## 2 Constraint Programming

The model shown is built using MiniZinc and ran using its interface on the Python programming language.
In addition to the variables given as input, there are also other useful variables that should be used:

1. A set of integers given by the numbers of couriers namely $VEHICLE$; a set of integers given by the number of items to be delivered $CUSTOMERS$; a set of integers in the range $0..\max(load)$, $LOAD$.
2. The set of nodes to be visited during the trip:
$$NODES = 1..items + 2 \cdot courier$$
Given by the fact that each courier should go depart and go back to the depot.
3. The set of depot nodes to be visited during the trip:
$$DEPOT\_NODES = items + 1..items + 2 \cdot courier$$
4. The set of depot nodes where couriers start the trips:
$$START\_DEPOT\_NODES = items + 1..items + courier$$
5. The set of depot nodes where couriers conclude the trips:
$$END\_DEPOT\_NODES = items + courier + 1..items + 2 \cdot courier$$

Since the purpose is to understand the whole path of the couriers at once, the weights of the items and the distance matrix must be adapted to the giant tour representation:

- An array of integers with length $NODES$:
$$demand_i = \begin{cases} weight_i & \text{if } i \leq items \\ 0 & \text{otherwise} \end{cases} \qquad \forall i \in NODES$$
- A matrix of integers with shape $NODES \times NODES$:
$$distance_{i,j} = \begin{cases} dist_{i,j} & \text{if } i \leq items \land j \leq items \\ dist_{items+1,i} & \text{if } i \leq items \land j > items \\ dist_{j,items+1} & \text{if } j \leq items \land i > items \\ dist_{items+1,items+1} & \text{otherwise} \end{cases} \qquad \forall i,j \in NODES$$

## 2.1 Decision variables

There are five decision variables in the model:

1. $successor, predecessor \in NODES$ that are arrays of length $NODES$ representing respectively the chosen path (where in each position $i$ there is its successor) and predecessors of each node (at each position $i$ there is the predecessor of the node $i$).
2. $vehicle \in VEHICLE$ (length $NODES$) that shows which vehicle visits which customer.
3. $intload \in LOAD$ (length $NODES$) that has the values of the loads when arriving at node $n \in NODES$.
4. A variable $tot$ representing the variable to be minimized as the objective.

## 2.2 Constraints

Constraints can be divided in section based upon which variable they will tackle. Some of them will be declared as redundant: in many cases adding constraints which are redundant, i.e. are logically implied by the existing model, may improve the search for solutions by making more information available to the solver earlier.

**Initialization constraints:**

- Successors of end nodes are start nodes:

$$successor_{items+2\cdot courier} = items + 1$$

$$successor_n = n - courier + 1 \qquad \forall n \in items + courier + 1..items + 2 \cdot courier - 1$$

- Associate each start/end nodes with a vehicle:

$$vehicle_n = n - items \qquad \forall n \in START\_DEPOT\_NODES$$

$$vehicle_n = n - items - courier \qquad \forall n \in END\_DEPOT\_NODES$$

- Vehicle load when starting at the depot (the problem remains essentially the same considering couriers picking up items from customers during the tour):

$$intload_n = 0 \qquad \forall n \in START\_DEPOT\_NODES$$

**Predecessor/successor constraints**:

- Initialization of *successor* and *predecessor*:

$$successor_{predecessor_n} = n \qquad \forall n \in NODES$$

$$predecessor_{successor_n} = n \qquad \forall n \in NODES$$

- All different and subtour elimination constraints given by:

$$circuit(successor) \qquad circuit(predecessor)$$

  The Gecode's global constraint *circuit* constrains the elements of $x$ to define a circuit where $x_i = j$ means that $j$ is the successor of $i$.

**Vehicle constraints**: vehicle of node $i$ is the same as the vehicle for the predecessor:

$$vehicle_{predecessor_n} = vehicle_n \qquad \forall n \in CUSTOMER$$

$$vehicle_{successor_n} = vehicle_n \qquad \forall n \in CUSTOMER$$

**Load constraints**:

- Load at item $n$ plus its weight is equal to the load at the successor of $n$:

$$intload_n + demand_n = intload_{successor_n} \qquad \forall n \in CUSTOMER$$

  At start nodes, there is no weight to be added:

$$intload_n = intload_{successor_n} \qquad \forall n \in START\_DEPOT\_NODES$$

- Check that partial load is always less or equal then the capacity for each vehicle:

$$intload_i \leq load_{vehicle_i} \qquad \forall i \in CUSTOMER$$

- Check that final load is less or equal then the capacity for each vehicle:

$$intload_{i+items+courier} \leq load_i \qquad \forall i \in VEHICLE$$

### 2.2.1 Symmetry breaking

Some symmetry breaking constraints were implemented in order to decrease the overall search space. Anyway, they are not used in the final model since they do not really speed up the search: on the contrary, the solution gets worse.

- **Courier with the same capacity are interchangeable**: every tour starts and ends a the same depot so couriers with the same capacity are completely interchangeable. It is possible to impose the following lexicographic constraint, thanks to the procedure $equality(a, b)$ which returns 1 if $a = b$, 0 otherwise.

$$\begin{aligned} load_i = load_j \wedge i > j \implies \text{lex\_less}(&[equality(vehicle_k, i) \mid \ k \ \in CUSTOMER\ ], \\ &[equality(vehicle_w, j) \mid \ w \ \in CUSTOMER\ ]) \end{aligned}$$

$$\forall i, j \in VEHICLE$$

- **Avoiding symmetric tour**: it is possible to force the solver to avoid symmetric tours of couriers (tours where couriers ship the same items in an inverse order) imposing the lexicographic ordering constraint between *successor* and *predecessor* decision variables:

  $$lex\_less([samecouriersucc_{i,c} \mid i \in \ NODES], [samecourierpred_{j,c} \mid j \in \ NODES])$$

  $$\forall c \in VEHICLE$$

  In fact, if *successor* and *predecessor* are equal in two different solutions, couriers deliver the same items but in an inverse order. The two functions $samecouriersucc_{i,c}$ and $samecourierpred_{i,c}$ return respectively the value of $successor_i$ and $predecessor_i$ if the package $i$ is taken by the courier $c$.

## 2.3 Objective

The variable to minimize is the one carrying the total distance of the tour, that is:

$$tot = \sum_{i \in NODES} distance_{i,successor_i}$$

Where $distance_{i,successor_i}$ is the Manhattan distance between $i$ and $successor_i$.

## 2.4 Search strategies

MiniZinc does not declare a specific search strategy to reach the solution. This leaves the search completely up to the underlying solver. Sometimes, particularly for combinatorial integer problems, in order to improve the performance it is useful to specify how the search should be undertaken. This requires us to communicate to the solver a search strategy.

Given the experiments, the best results are obtained using concatenated integer searches of the decision variables inside a sequential search. Moreover, *Dom_W_Deg* search strategy has been used for the *successor* and *predecessor* decision variables because they are strictly correlated to the objective. This strategy weights the importance of a variable based on how many times it has been in a constraint that caused failure earlier in the search.
Then, $first\_fail$ has been used for the *vehicle* and the *int_load* decision variables in order to have a reliable assignment of the items to the couriers early in the search, since it searches first for the correct value for the given variables.
Experimenting with *indomain_median*, *indomain_split*, *indomain_min* and *indomain_random* resulted in the choice of the last one, so to have a different search at each restart.
In fact, any kind of depth first search for solving optimization problems suffers from the problem that wrong decisions made at the top of the search tree can take an exponential amount of time to undo. One common way to ameliorate this problem is to restart the search from the top thus having a chance to make different decisions.
Different restart annotations control how frequently a restart occurs. Restarts occur when a limit in nodes is reached, where search returns to the top of the search tree and begins again.

*Restart_luby* seems to be the most effective restart, therefore is used in our final version. It gets as input a variable *scale*: the $k$-th restart gets $scale \cdot L_k$ where $L_k$ is the $k$-th number in the Luby sequence. The Luby sequence repeats two copies of the sequence ending in $2^i$ before adding the number $2^{i+1}$ (Luby sequence: 112112411211248...). It means that there is a correlation between the number of restarts occurred and the nodes limit of the search (the later the restart, the higher the limit of nodes).
Similar results are also performed by *restart_linear* where *scale* parameter defines the initial number of nodes before the first restart. The second restart gets twice as many nodes, the third gets three times, etc.

Gecode solver also provides the *relax_and_reconstruct* annotation that is a simple large neighbourhood search strategy: upon restart, for each variable in $x$, the probability of it being fixed to the previous solution is percentage (out of 100). For each instance, percentage selected may vary the result.

Anyway, while handling small and easy instances like the example one, these techniques should be avoided since they could make the search more difficult for the solver.

## 3 Satisfiability Modulo Theories (SMT)

The model shown is built using Z3Py, the Z3 API in Python. It takes as inputs:

1. *num_items*: the number of items to be delivered.
2. *num_couriers*: the number of couriers available.
3. *items_weights*: an array containing the weights of each items.
4. *courier_loads*: an array containing the load of each couriers.
5. *dist*: a matrix containing in the position $i, j$ the distance between the item $i$ and the item $j$; the last row and column of the matrix contain the distance from the depot.

### 3.1 Decision variables

There are three decision variables in the model:

1. *track*: a three dimensional Boolean matrix, representing a directed Graph.

   $$track_{i,j,c} = True$$

   Means that the courier $c$ moves from $i$ to $j$, where $i$ and $j$ can represent both an item or the depot.

2. $u$: an additional array to implement the Miller-Tucker-Zemlin formulation for subtours elimination.

3. *tot*: contains the sum of the distances traveled by all the couriers, so the objective that the solver has to minimize.

### 3.2 Predicates

For a better understanding of the model, assume that the Boolean values of the matrix track $True$ and $False$ are replaced by 1 and 0 respectively. The Z3 solver seems to have better performance on Boolean variables and Z3py has simple instructions to treat $True$ and $False$ as 0 and 1.

- The main diagonal has to be False for each courier

$$track_{i,i,c} = 0 \qquad \forall i \in 0..num\_items,\ \forall c \in 0..num\_couriers - 1$$

- Each node has to be visited only once: so in every row and columns has to appear only one True so that every items is reached and leaved.

$$\sum_{i=0}^{num\_items} track_{i,j,c} = 1 \ \wedge \ \sum_{i=0}^{num\_items} track_{j,i,c} = 1$$

$$\forall j \in 0...num\_items - 1,\ \forall c \in\ 0...num\_couriers - 1$$

- Each courier can depart from depot and return to depot only once:

$$\sum_{j=0}^{num\_items-1} track_{num\_items,j,c} = 1 \quad \wedge \quad \sum_{j=0}^{num\_items-1} track_{j,num\_items,c} = 1$$

$$\forall c \in 0..num\_couriers - 1$$

- This constraint ensures that the load of each couriers is not exceeded.

$$courier\_load_c \geq \sum_{\substack{i \in 0...num\_items-1 \\ j \in 0...num\_items}} items\_weights_i \cdot track_{i,j,c} \qquad \forall c \in \ 0..num\_couriers - 1$$

- Every courier has to leave all the nodes that he reaches, depot included

$$\sum_{i=0}^{num\_items} track_{i,j,c} = \sum_{i=0}^{num\_items} track_{j,i,c}$$

$$\forall c \in 0...num\_couriers - 1, \ \ \forall j \in 0...num\_items$$

- **Miller-Tucker-Zemlin formulation for subtour elimination:**

$$u_i + track_{i,j,c} \leq u_j + \ num\_items \cdot (1 - track_{i,j,c})$$

$$u_i \geq 0$$

$$\forall c \in 0...num\_couriers - 1, \ \ \forall i,j \in 0...num\_items - 1$$

## 3.3 Objective

The Optimizer has to minimize the decision variable *tot* representing the total distance:

$$tot = \sum_{i,j,c=0}^{\substack{i,j=num\_items \\ c=num\_couriers-1}} track_{i,j,c} \cdot dist_{i,j}$$

## 3.4 Subtour elimination: two alternatives

**Miller-Tucker-Zemlin formulation** for subtour elimination needs an additional variable, called $u$, that gets a value for each node, except for the depot. If a vehicle drives from node $i$ to node $j$, the value of $u_j$ has to be bigger than the value of $u_i$. Therefore a courier can not visit a node twice in his path because the visited node already has a lower value of $u$ with respect to the current one. This also ensures that a vehicle will not drive in a circle.
The depot does not have a value of $u_i$, because it is mandatory for the courier to visit the depot twice, since the vehicle starts and ends at the depot.

Another possible formulation is the **Dantzig-Fulkerson-Johnson** formulation. For each subset of nodes (depot excluded) there must be at least two edges between the nodes in S and in its complementary. In this way, cycles not including the depot are avoided.

$$\sum_{\substack{i \in S \\ j \notin S}} track_{i,j,c} \geq 2 \qquad S \subseteq \{0...num\_items - 1\} \qquad 2 \leq \mid S \mid \leq num\_items - 2$$

## 3.5 Issues

Adding all the constraint for subtour elimination may take a lot of time for the big instances, and unfortunately Z3py Optimizer does not allow to add these constraint in a lazy way.
It is possible to add the subtour elimination to the model in a lazy way proceeding as follows. The Optimizer has to be replaced by the Solver and each time a solution is found:

- If the solution contains subtours:
  1. Save the subsets of the subtours as $S_i$
  2. Eliminate the specific subtour S adding either:
     * $\sum_{\substack{i \in S \\ j \notin S}} track_{i,j,c} \geq 2$ for DFJ.
     * $\substack{u_i + track_{i,k,c} \leq u_k + num\_items \cdot track_{i,k,c} \\ \forall i \in S,\ \forall k \in 0..num\_items-1,\ \forall c \in 0..num\_couriers-1}$ for MTZ.
- If the solution does not contains subtours:
  1. Save the value of the objective $objective_i$
  2. Force the Solver to find a solution with a lower value of the objective, adding

$$tot < objective_i$$

This method saves a lot of the time needed to add the subtour elimination to the model but the Solver is not meant to look for the lowest objective so its value decreases really slowly.

# 4 Mixed-Integer Linear Programming (MIP)

Concerning the MIP formulation, our choice was to use the Gurobi solver and in particular its interface on the Python programming language.

## 4.1 Decision variables

Given a number of items to be delivered $N$, a number of available couriers $K$, a set of vertices connecting each item plus a dummy node 0 (the depot) $V$, the problem is represented using a set of Boolean variables:

$$x_{i,j,k} \qquad i, j \in V,\ i \neq j,\ k \in K$$

When the element $x_{i,j,k}$ is 1, it means that there is an active arc that connects the items $i$ and $j$, delivered by the courier $k$.

## 4.2 Constraints

The first constraints for the model are the ones that assures that each customer are visited only once:

$$\sum_{k \in K} \sum_{i \in V} x_{i,j,k} = 1 \qquad \forall j \in N,\ i \neq j$$

$$\sum_{k \in K} \sum_{j \in V} x_{i,j,k} = 1 \qquad \forall i \in N,\ i \neq j$$

The second ones avoid that couriers can leave and enter the depot multiple times:

$$\sum_{j \in N} x_{0,j,k} = 1 \qquad \forall k \in K$$

$$\sum_{j \in N} x_{j,0,k} = 1 \qquad \forall k \in K$$

Then, the number of arcs that enters a node and the number of arcs that leaves the same node are equal.

$$\sum_{i \in V} x_{i,j,k} - \sum_{i \in V} x_{j,i,k} = 0 \qquad \forall j \in V,\ i \neq j,\ \forall k \in K$$

Finally, the load constraint over all the couriers:

$$\sum_{i \in V} \sum_{j \in N,\ j \neq i} x_{i,j,k} \cdot q_j \leq C_k \qquad \forall k \in K$$

Where $q_j$ is the weight of the item $j$ and $C_k$ is the total load of the courier $k$.

### 4.2.1 Subtour elimination techniques

In order to get better results, two techniques have been implemented:

- **Miller-Tucker-Zemlin (MTZ) formulation**: same formulation as SMT but with a slight difference since it also takes into account the partial load of each courier.

$$x_{i,j,k} = 1 \implies u_i + q_j = u_j \qquad \forall i \in N,\ \forall j \in N,\ i \neq j,\ \forall k \in K$$

$$x_{i,j,k} = 1 \implies u_i \leq C_k \qquad \forall i \in N,\ \forall j \in V,\ i \neq j,\ \forall k \in K$$

$$q_i \leq u_i \qquad \forall i \in N$$

  Couriers' loads are different, so the indicator constraints are needed in order to considerate only subsets of $u_i$ (one subset for each courier).

- **Generalized Subtour Elimination Constraints (GSEC)**: it is a group of constraints that eliminate subtours as for every proper subset of nodes $S_k \subseteq V$, for each $k \in K$, it forbids the number of selected arcs within $S_k$ to be equal to or larger than the number of nodes in $S_k$.
  The number of constraints of the model is exponential. For this reason, the model cannot be used directly, unless the number of nodes is small. The solution is to add to the model only those subtour elimination constraints that are really needed, by means of a **lazy approach**, just as reported in SMT.

$$\sum_{i,j \in S_k} x_{i,j,k} \leq |S_k| - 1 \qquad \forall k \in K,\ S_k \subseteq V$$

  In Gurobi, a lazy approach is implemented using callbacks.

## 4.3 Objective

The objective is to minimize the total distance traveled by all the couriers, therefore:

$$\sum_{i \in V} \sum_{j \in V,\ j \neq i} \sum_{k \in K} x_{i,j,k} \cdot c_{i,j}$$

Where $c_{i,j}$ is the Manhattan distance between items $i$ and $j$ (the length of the arc connecting them).

## 4.4 Warm start

A useful technique which can help to improve the initial solution of the solver and speed up the computation is the warm start or MIP start. Basically, it consists in providing an initial solution to the decision variables.
We implemented a function that creates an initial solution providing to the bigger couriers the farthest node as first visit and then a series of nodes which are close together near the first one. Efficiency is given by the fact that bigger couriers could travel more distance than smaller ones if

the items carried that are far from each others are also lighter.
Anyway, since it does not take into account the partial load of a courier completely, it could have problems with the last item (last free place could be split among couriers). If so, another kind of start is implemented: a function fills the couriers (sorted in a decreasing load order) with packages (decreasing weight order) until it reaches a solution satisfying the load constraint.

In general, anyway, providing a MIP start could be misleading for the solver as it would focus the search only in a subset of the search space. This effect is deleterious to the optimality of the search if the MIP start is not very similar to the best route the couriers could take.

## 5 Comparing the results

| – | inst01 | inst02 | inst03 | inst04 | inst05 | inst06 | inst07 | inst08 | inst09 | inst10 | inst11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| CP | 1162 | 1976 | 4042 | 3364 | — | — | 1460 | 2750 | 4612 | 15866 | 1168 |
| SMT | 2202 | — | — | — | — | — | 2660 | — | — | — | 2166 |
| MIP$_{GSEC}$ | 1144 | 2976 | 14488 | 16968 | 19078 | 25478 | 1692 | 13928 | 20686 | 26610 | 1150 |
| MIP$_{MTZ}$ | 1184 | 2518 | 12360 | 14950 | 17518 | 25478 | 1594 | 5088 | 19520 | 26610 | 1160 |

Table 1: Results comparison (red - best solutions; blue - warm starts)

### 5.1 Comments

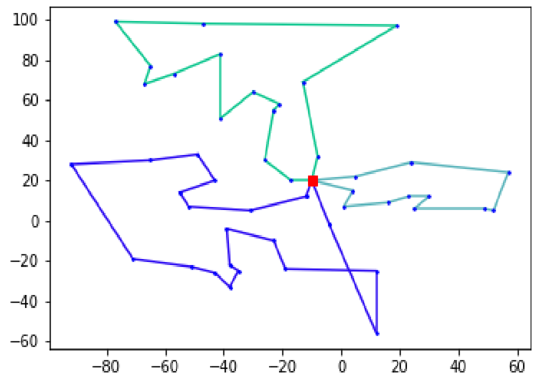
The **CP** solvers achieves good results in all the instances where is able to find a correct solution. In fact, Gecode annotation *relax_and_reconstruct* helps to find solution similar to the previous one. A slightly change on the probability that a variable $x$ will take the same value as the previous solution strongly affects the results (we obtained good results setting this percentage between 89 and 92). So a solution can be easily improved once a solution is found, but using a value of *relax_and_reconstruct* too high may lead to local minima.
As for the instances 05 and 06, it would be possible to try giving an initial solution using the warm start, but it was not implemented since it would make our solution less general.

The **MIP** implementation using **GSEC** has outstanding results on small instances, but struggles to find results on bigger instances because of the lazy approach: in fact, while it solves the problem of the exponential number of constraints to be generated with an eager method, initially the solution space is too big to find a suitable solution in a reasonable time.
The **MTZ** formulation, instead, can find feasible solutions also in big instances, but the search is really slow and results are really big if the timeout interval is small. Opposed to the lazy approach, this formulation will run out-of-memory when solving instances 06 and 10 because of the huge amount of memory needed for the search.
Concerning the warm start, it is helpful to find better solutions in all the instances compared to the default search, but it is also the reason why bigger instances are less prone to an improvement.

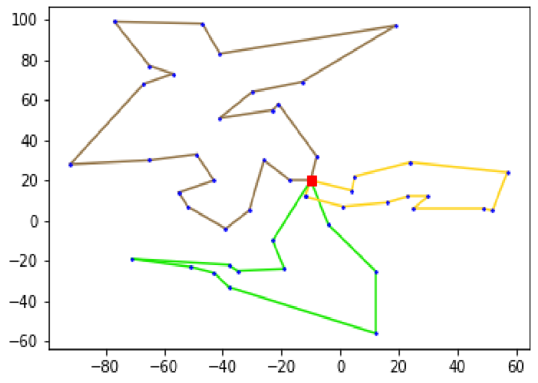
**SMT** results on the table are the ones of the model using Miller-Tucker-Zemlin (not in a lazy way): it provides some good results in the smaller instances, but it barely has the time to define the model for the big ones and sometimes it also runs out-of memory. On the other hand, the lazy approach compiles the model in few seconds but it does not work well as it comes to minimize the objective due to the huge search space.
In order to reduce it, a model that took advantage of the symmetry, using an undirected graph, was implemented. Yet, more constraints than the asymmetric model had to be used, therefore it was performing far worse even if it was handling fewer decision variables.

## 5.2 Plots

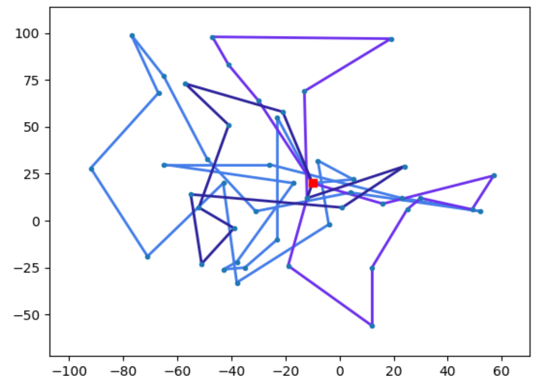
In order to further understand the behavior of the different models, the plot could be useful. The travelled route of each courier is reported in the folder *out* of each model.



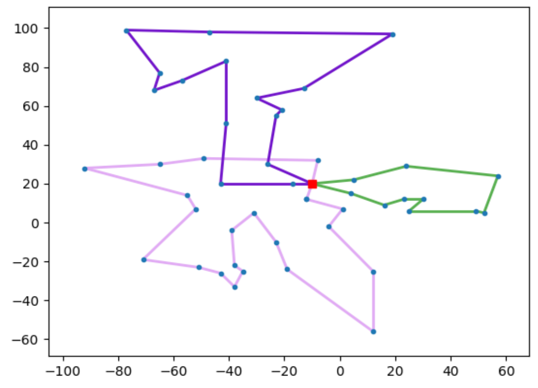
(a) MIP$_{GSEC}$



(b) MIP$_{MTZ}$

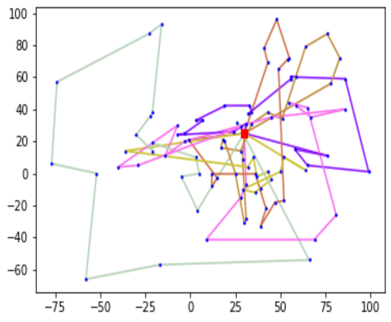


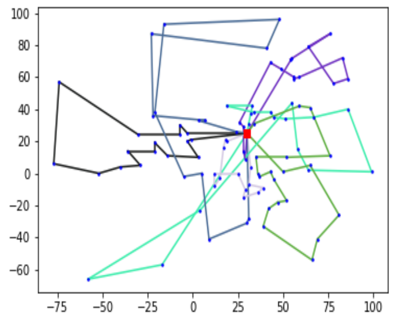
(c) SMT



(d) CP

Figure 1: Instance 01


(a) MIP$_{GSEC}$


(b) MIP$_{MTZ}$

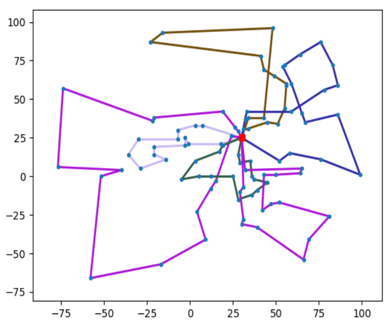
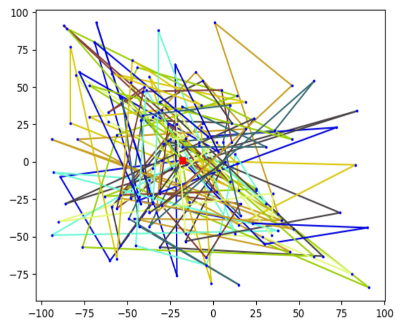
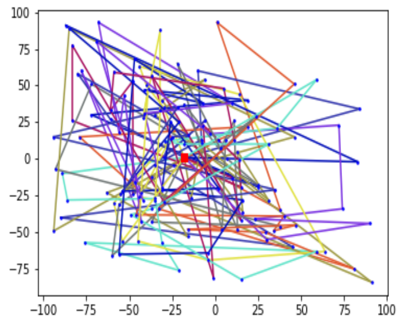

(c) CP

Figure 2: Instance 02


(a) MIP$_{GSEC}$


(b) MIP$_{MTZ}$

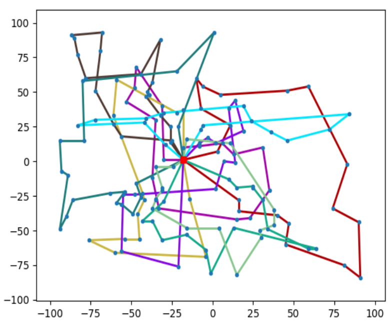
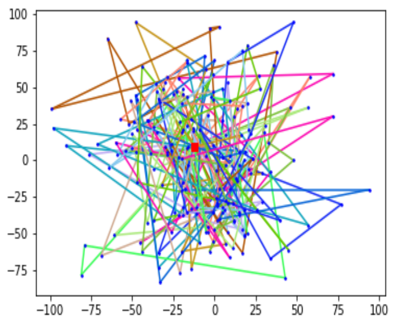
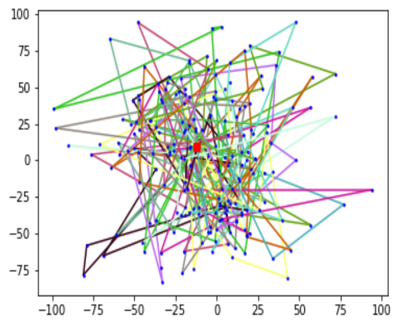

(c) CP

Figure 3: Instance 03


(a) MIP$_{GSEC}$


(b) MIP$_{MTZ}$

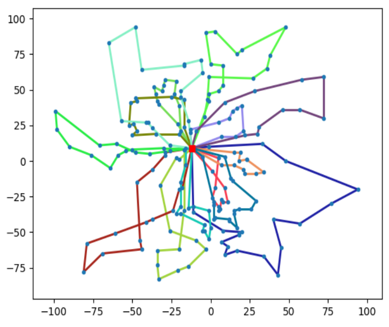
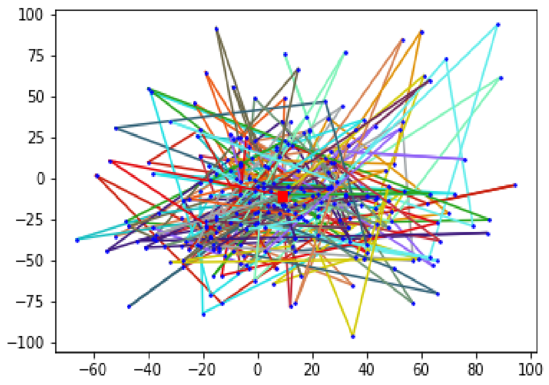
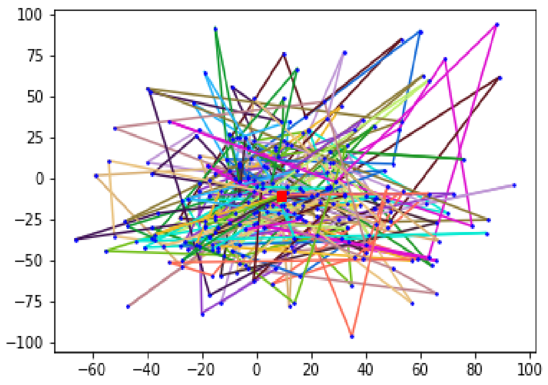

(c) CP

Figure 4: Instance 04

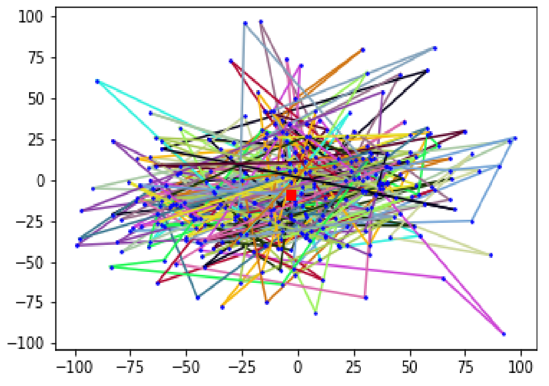


(a) MIP$_{GSEC}$

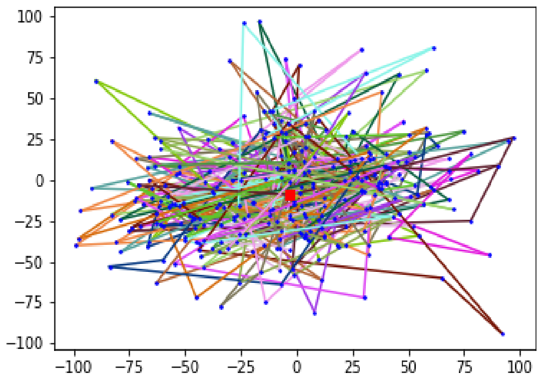


(b) MIP$_{MTZ}$

Figure 5: Instance 05

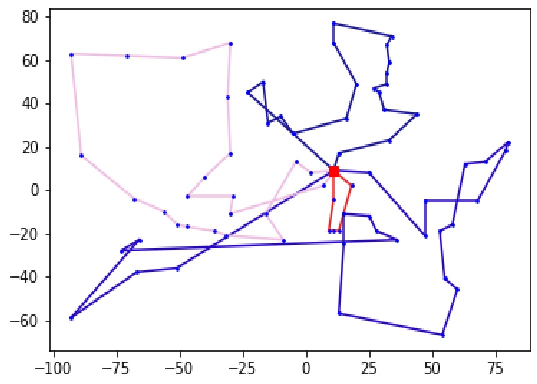


(a) MIP$_{GSEC}$

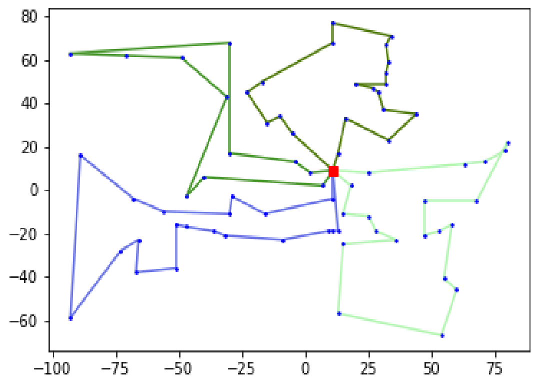


(b) MIP$_{MTZ}$

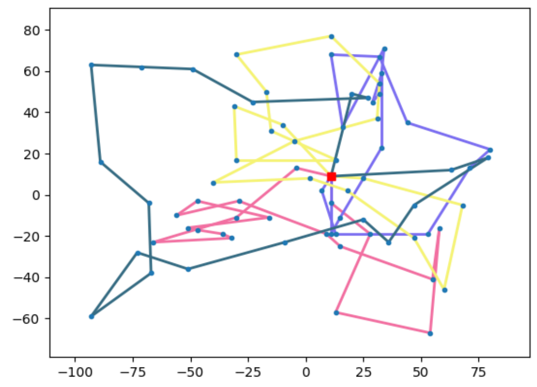
Figure 6: Instance 06



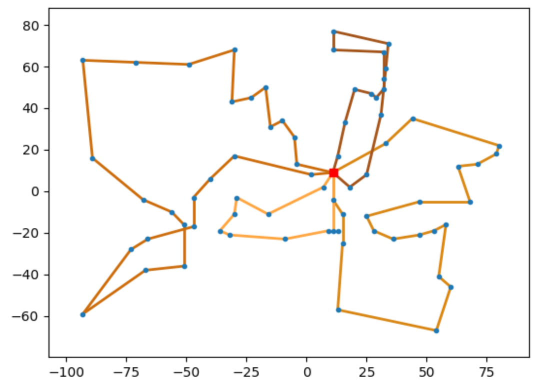
(a) MIP$_{GSEC}$



(b) MIP$_{MTZ}$

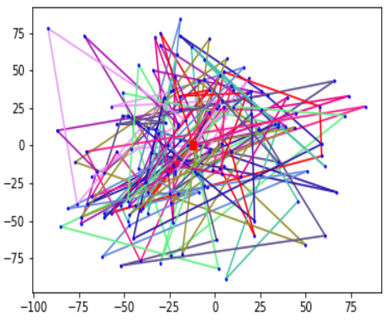


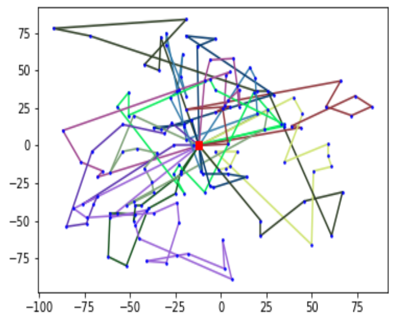
(c) SMT



(d) CP

Figure 7: Instance 07


(a) MIP$_{GSEC}$


(b) MIP$_{MTZ}$

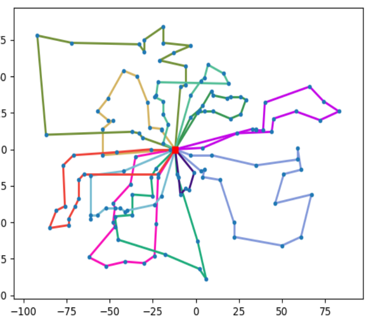
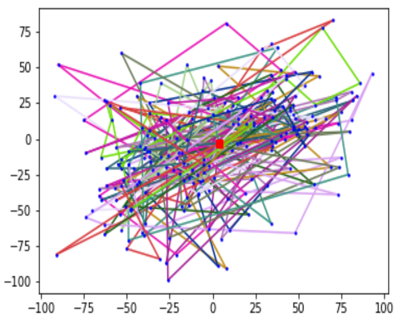
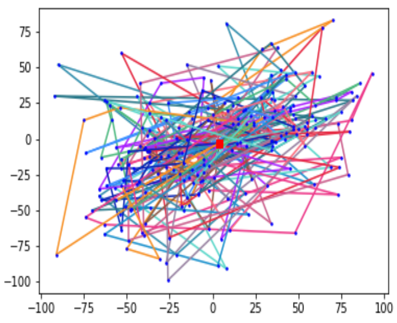

(c) CP

Figure 8: Instance 08


(a) MIP$_{GSEC}$


(b) MIP$_{MTZ}$

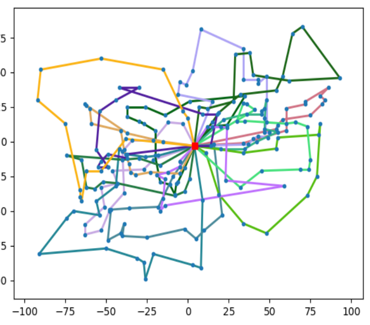
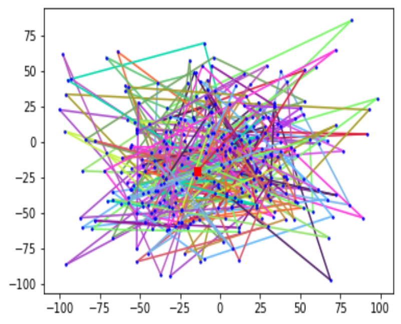
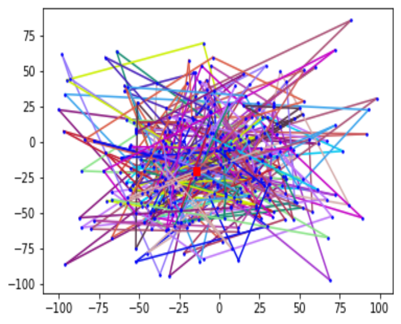

(c) CP

Figure 9: Instance 09


(a) MIP$_{GSEC}$


(b) MIP$_{MTZ}$

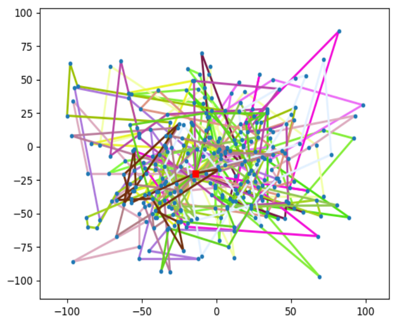
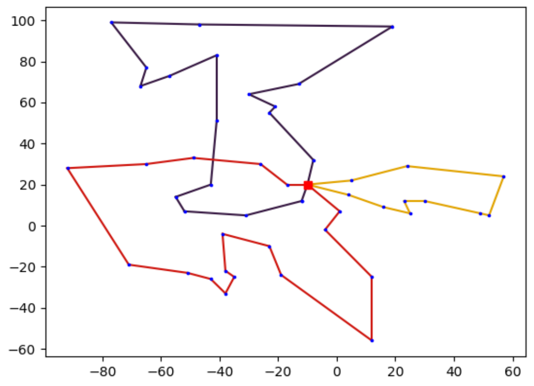
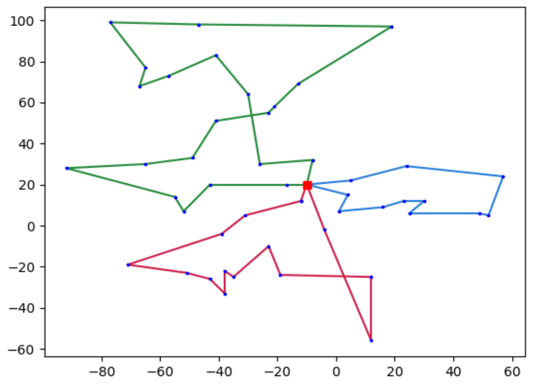

(c) CP

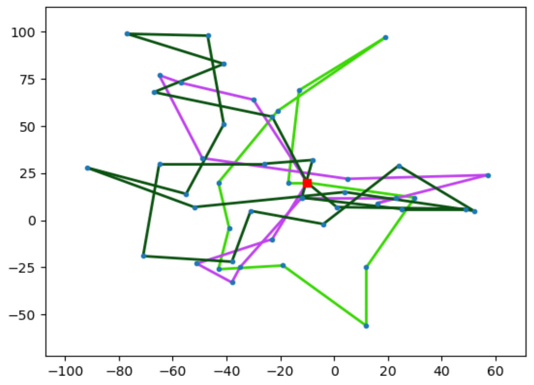
Figure 10: Instance 10



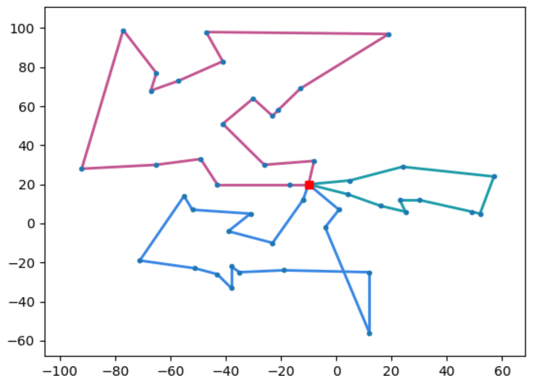
(a) MIP$_{GSEC}$



(b) MIP$_{MTZ}$



(c) SMT



(d) CP

Figure 11: Instance 11

## 6 Conclusions

In this project, we have implemented the Multiple Courier Planning or, using its literature name, the Vehicle Routing Problem, which is a generalization of Travelling Salesman Problem.
We have been able to use both Contraint Programming, Satifiability Modulo Theories and Linear Programming to find suitable solutions for a set of instances with sizes ranging from small to very big. At the end, the best results were given by the first paradigm, but using a better performing machine could have given outstanding results in all of the implementations.
Anyway, since this problem is one among the most classic in Combinatorial Optimization, there are plenty of different approaches that could have been studied.

Reading the literature, we found some hybrid approaches. These kinds of algorithms basically try to combine basic heuristic methods in higher level frameworks in order to efficiently and effectively explore the search space. They are also commonly called meta-heuristics. Meta-heuristic approaches are very efficient for escaping from local optimum. A great number of meta-heuristic algorithms have been applied to the VRP and they have achieved great performances (i.e, Honey Bee Mating Optimization or Ant Colony Optimization).
The results of these approaches seem to be good in both small and huge instances. This could have been the ideal choice for an improvement of our models and, despite the better performance that a specific model might have, an hybrid approach is usually more versatile.
Most hybrid models are proprietary, but there are plenty that are open source: in the future they could be the stepping point to implement our own improved solution.

Otherwise, another improvement could have been simply given by decreasing the input information, through the creation of a correlation between very near customers (or equally, items) so that packages with less distance between each others have an higher priority to be brought all together by the same courier.

## References

[1] Vehicle Routing Problem, `https://en.wikipedia.org/wiki/Vehicle_routing_problem`, *Wikipedia*

[2] An analytical bound on the fleet size in vehicle routing problems: a dynamic programming approach, `https://arxiv.org/pdf/1905.05557.pdf`, *arXiv*

[3] Miller-Tucker-Zemlin, `https://how-to.aimms.com/Articles/332/332-Miller-Tucker-Zemlin-formulation.html`, *AIMMS*

[4] Gurobi documentation, `https://www.gurobi.com/documentation/`, *Gurobi*

[5] MiniZinc documentation, `https://www.minizinc.org/doc-2.6.4/en/part_3_user_manual.html`, *MiniZinc*

[6] MiniZinc benchmarks, `https://github.com/MiniZinc/minizinc-benchmarks`, *MiniZinc*

[7] Z3 documentation, `https://ericpony.github.io/z3py-tutorial/guide-examples.htm`, *Z3*

[8] Models, relaxations and exact approaches for the capacitated vehicle routing problem, `https://mate.unipv.it/~gualandi/famo2conti/papers/routing_models.pdf`, *UniPD*

[9] A hybrid approach for the vehicle routing problem with three-dimensional loading constraints, `https://www.sciencedirect.com/science/article/pii/S0305054811003388`, *ScienceDirect*

[10] Solving the vehicle routing problem by a hybrid meta-heuristic algorithm, `https://link.springer.com/article/10.1186/2251-712X-8-11`, *Springer*