Multiple Couriers Planning

Riccardo Paolini (riccardo.paolini5@studio.unibo.it),
Enrico Mannocci (enrico.mannocci@studio.unibo.it),
Matteo Periani (matteo.periani2@studio.unibo.it)

July 2022

Contents

1 Introduction

In this report we will show our solution to the Multiple Couriers Planning Problem. We have built three different model respectively a Constraint Programming one (*CP_Or_Tools.py*), a Linear Programming one (*LP_Gurobi.py*) and an SMT one (*SMT_Z3.py*). The task that was given to us contains 11 instances that we had to solve, we will print our solutions in the *Result* sections. Before running our model we have preprocessed the data of the given instances (*instGen.py*).

2 Preprocessing & Simplifications

In this paragraph we will highlight the preprocessing of the instances. In fact, before diving into the problem, we have extracted some useful information from the data, such as the calculation of distances, the lower bound, the reduction of couriers used and we have created clusters of items in the bigger instances.

2.1 Distribution Points

Since the whole problem is based on reducing the total distance travelled by our couriers, we need to know the distances between different Distribution Points.

In particular, we decided to consider the **Manhattan distance**:

$$dist_{s \to d} = |x_s - x_d| + |y_s - y_d|$$

where **s** is the starting point, and **d** is the destination point. So, we have created a **Distance matrix, D**, in which the distances between each pairs of points are stored.

	1	2	3	D
1	0	7	4	13
2	7	0	9	5
3	4	9	0	1
D	13	5	1	0

Table 1: Distance matrix

2.2 Lower Bound

The evaluation of a **Lower Bound, LB**, is a necessary step in order to reduce the *Search Space* and find the optimal solution. However, finding a good lower bound was not easy.

Our first attempt was based on taking the summation of the minimum distances (excluding the diagonal) for each column of the **Distance matrix**.

	1	2	3	D
1	-	7	4	13
2	7	-	9	5
3	4	9	-	1
D	13	5	1	-

Table 2: Distance matrix

In this case the Lower Bound will be evaluated as follows:

$$LB = 4 + 5 + 1 + 1$$

This lower bound is correct given that we are adding together the shortest segments we have in the map. On the other hand, we are not satisfied with this choice and we have opted for something more complicated.

Thus, we found a better lower bound by removing a vertex and finding a Minimum Spanning Tree using the Prim's Algorithm as reported here. Finally, we added the lengths of the two shortest edges connected to the missing vertex. The lower bound we obtained gives us a lot of information as it is much tighter than the trivial one.

2.3 Reduction of Couriers

Given the size of the instances to solve, we tried to reduce their complexity by using some math. For instance, *Triangle Inequality* tells us that the distance to go from point **A** to point **B** is certainly less than or equal to the distance necessary to go from **A** to **B** passing through **C**. We exploited this fact to reduce the number of couriers to the minimum number that can transport all the objects. Indeed, if we have an object which is not yet delivered, it must be allocated to one of the couriers that are already partially filled, otherwise we are ensured to follow a longer path and we would not get an optimal solution.

e.g. instance 2: 20 couriers $\rightarrow$ 6 couriers

2.4 Clustering of Items

To obtain some results in the hardest instances we have reduced the number of items aggregating the closer ones. We transformed these clusters of points into new single items with coordinates given by the mean of the points and the weight equal to the sum of the weights.

We have chosen 40 as maximum number of points. Therefore, in all the instances that have more than this limit, we aggregate items together until we reach the chosen number, e.g. 40. This parameter is very important because with too many points we would have a useless simplification, moreover some

instances are too complicated to find even a bad solution. On the other hand, we may get an unfeasible model if we reduce it too much.

When aggregating points it is important not to have an object that is too heavy as it can lead to an unfeasible model, so we have placed a limit on it:

$$w_1 + w_2 \leq max_{load} \tag{1}$$

where w_1 and w_2 are the weights of the items we want to aggregate and max_{load} is the maximum load between couriers. We aggregate only points that satisfy (1).

As an example there we have a solution for a simplified instance:

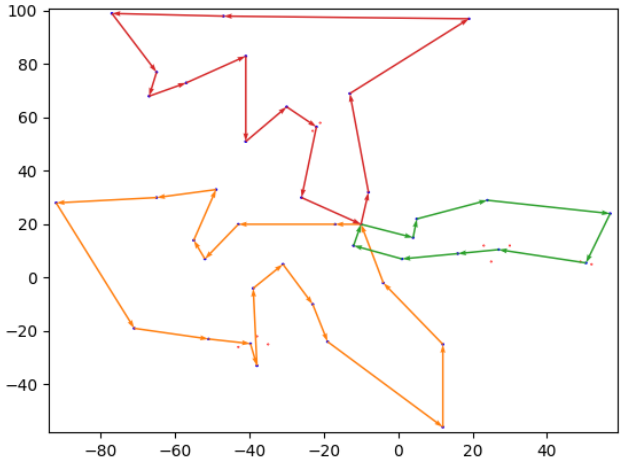


Figure 1: With arrows of different colors we define different couriers, blue points are single item or centers of clusters. Orange points are element inside a cluster.

This is, of course, **not** a final solution but just a partial one. To obtain a final solution we have run a smaller model for each courier. In those models we have optimized just the order at witch all items are collected keeping fixed the items that belongs to each courier.

3 Constraint Programming Model

To solve this problem instance we have used OR-Tools: a Google developed Constraint Programming Solver.

3.1 Problem Representation

We have represented the problem as a 3D Boolean Matrix $m \times N \times N$ (*table*) where m is the number of couriers and N is the number of items plus one ($N = n + 1$). Where the *plus one* is for the Depot representation.

The fact that the element $table_{k,i,j}$ is equal to 1 means that the k^{th} courier travels from the i^{th} to the j^{th} item.

	1	2	3	D
1	0	0	0	0
2	0	0	1	0
3	0	0	0	1
D	0	1	0	0

Table 3: Example of a courier carrying 3 items. The courier travel will be: Depot → second-item → third-item → Depot

Then, we filled the matrix according to the problem description. To this end, we must be sure that:

- each row/column contains exactly a 1 (considering all the courier matrices as if they were summed - no diagonal): in this way **each item is delivered exactly once**.
- **weight satisfaction**: do not exceed the courier's capacity.
- no sub-loops within matrices: ensures that each courier perform an ***Hamiltonian path***.

	1	2	3	4	D
1	0	1	0	0	0
2	0	0	1	0	0
3	1	0	0	0	0
4	0	0	0	0	1
D	0	0	0	1	0

Table 4: Example of a possible loop due to the incomplete definition of the constraints (circuit constraint is missing).

To deal with this huge problem, we have used the **Circuit** constraint provided by OR-Tools. It will be explained in the following section.

3.2 Constraints

In this section we will list the constraints we used.

The first constraint is **Exactly One** that OR-Tools translates into a sum of the rows/columns and makes sure it equals 1. We have used it to ensure that each *Delivery Point* is first reached and then left, plus it is delivered by only one courier in the fleet.

$$ExactlyOne(\forall k \, \forall j_{\neq i} \; x_{k,i,j}) \quad \forall \, i = 1..N \tag{2}$$

$$ExactlyOne(\forall k \, \forall i_{\neq j} \; x_{k,i,j}) \quad \forall \, j = 1..N \tag{3}$$

The fact of excluding the diagonal is due to a tricky implementation of the Circuit constraint, that forces each matrix to have a 1 in every row. However, this is not a problem, because we just have to put a 1 in those diagonal cells that, in any case, would not have been used by the model.

Then, the bottom right cell of each matrix is set to 0, this ensures that every courier leaves the depot. In fact, as we said in the section 2.3, we have already reduced the number of couriers to the minimum.

$$x_{k,N,N} = 0 \qquad \forall \, k = 1..m \tag{4}$$

where m is the number of couriers.

At this point we can observe the constraint that regulates the weights loaded on each courier. This prevents the total load from exceeding the courier's limit.

$$\sum_{i=1}^{N} \sum_{j=1}^{N-1} x_{k,i,j} * s_j \leq load_k \qquad \forall \, k = 1..m \tag{5}$$

where s represent the weight of the j^{th} object. We emphasize that, being the N^{th} column the one representing the depot, we do not need extra capacity to go there, this justifies the fact that j ranges from 1 to $N - 1$.

Finally, we have the most important and complex constraint, **Circuit**. It ensures the non-existence of sub-loops within the matrix, and it does so by forcing the arcs to create one single loop (*Hamiltonian path*).

$$Circuit(X_k) \qquad \forall \, k = 1..m \tag{6}$$

e.g. Circuit({(1, 2), (2, 4), (4, 1)}) is True

e.g. Circuit({(1, 2), (2, 3), (3, 2)}) is False

3.3 Objective Function

To get the objective function, we summed the distances traveled by all the couriers.

$$\text{OBJ} = \sum_{k=1}^{m} \sum_{i=1}^{n+m} \sum_{j=1}^{n+m} x_{k,i,j} \cdot D_{i,j} \tag{7}$$

where again $x_{k,i,j}$ is the k, i, j element of the *table* matrix and $D_{i,j}$ is the distance between the i-item and the j-item.

3.4 Solver

As first attempt, we tried to solve this problem by using the **Gecode** solver available on **Minizinc**. After obtaining poor results (in terms of distance traveled), we moved to **OR-Tools** solver which is one of the best performing solvers according to the MiniZinc Challenge 2021.

The following table shows the results of the two solvers on the 1^{st} instance with the same model implementation.

Solver	Time	Length
Gecode	5min	2872
OR-Tools	5min	1144

Table 5: Comparing Gecode and OR-Tools solvers.

4 Linear Programming Model

For the creation of our Linear Programming Model we have used a solver called Gurobi, it is a commercial product but we have had a 1 month licence thanks to our University.

4.1 Problem Representation

As for CP, we have represented the problem as a vector of m matrices (*table*) of size N×N, where m is the number of couriers and N is the number of items plus the depot ($N = n + 1$). In addition, we have kept the same logic used for CP. Therefore, when the element $table_{k,i,j}$ equals 1 means that the k^{th} courier travels from the i^{th} to the j^{th} item, where i and j are delivery locations.

Furthermore, the observations made for the CP model are still valid, so:

- each item can only be delivered by one courier;
- each item must be delivered;

- each courier has to start form the depot and return there at the end of the delivery path;
- avoid sub-paths and loops.

4.2 Constraints

In this section we will list the constraints we used to create the **Gurobi model**.

The first constraint sets to 1 the sum of each column (excluding the last one: depot), considering all the couriers' matrices combined. We used it to ensure that each *Delivery Point* is reached by exactly one courier (8). Then, we integrated it with another constraint that forces the courier to leave the *Delivery Point* if it has previously reached it (9).

$$\sum_{k=1}^{m}\sum_{i=1}^{N} x_{k,i,j} = 1 \qquad \forall\, j = 1..N-1 \tag{8}$$

$$\sum_{j=1}^{N} x_{k,i,j} = \sum_{j=1}^{N} x_{k,j,i} \qquad \forall\, k = 1..m \quad \forall\, i = 1..N \tag{9}$$

In addition to that, we have added a constraint that forces the diagonal values to be zero. We do not want a courier to loop in a *Delivery Point*.

$$x_{k,i,i} = 0 \qquad \forall\, k = 1..m \quad \forall\, i = 1..N \tag{10}$$

where m is the number of couriers.

Now, we introduce the constraints that let the courier move from the depot (11) and come back to it at the end of the delivery path (12).

$$\sum_{j=1}^{N} x_{k,N-1,j} = 1 \qquad \forall\, k = 1..m \tag{11}$$

$$\sum_{i=1}^{N} x_{k,i,N-1} = 1 \qquad \forall\, k = 1..m \tag{12}$$

Then, we have the constraint that regulates the weights loaded on each courier. This prevents the total load from exceeding the courier's limit.

$$\sum_{i=1}^{N}\sum_{j=1}^{N-1} x_{k,i,j} \cdot s_j \leq load_k \qquad \forall\, k = 1..m \tag{13}$$

where s represent the weight of the j^{th} object. We emphasize that, being the N^{th} column the one representing the depot, we do not need extra capacity to go there, this justifies the fact that j ranges from 1 to $N-1$.

The approach we used to substitute the **Circuit** constraint used in CP is the following. We start by creating N auxiliary variables, $u_1 \dots u_N$, among which the last one has value 1, $u_N = 1$, and the values of the remaining variables depend on the arcs selected. In particular, whenever an arc among the matrices is selected, e.g. $x_{k,i,j}$, it enforces $u_j > u_i$.

$$x_{k,i,j} * u_j \geq x_{k,i,j} * (1 + u_i) \qquad \forall\, k = 1..m \quad \forall\, i = 1..N \quad \forall\, j = 1..N \tag{14}$$

4.3 Objective Function

To get the objective function, we summed the distances traveled by all the couriers.

$$\text{OBJ} = \sum_{k=1}^{m} \sum_{i=1}^{n+m} \sum_{j=1}^{n+m} x_{k,i,j} \cdot D_{i,j} \tag{15}$$

where again $x_{k,i,j}$ is the k, i, j element of the *table* matrix and $D_{i,j}$ is the distance between the i-item and the j-item.

5 SMT Model

For the creation of our SMT model we have used the Python API of Z3, an open source solver. The choice to use this tool is given by the impossibility to use SMT-LIB due to the size of the file of the encoding. Indeed, when we were creating them, they reached dozens of megabits in size and the computation did not gave any result within the time limit.

5.1 Problem Representation

The representation of this problem is slightly different respect to the other ones. We have a 2D matrix of size $(n+m) \times (n+m)$ (*table*), a 2D matrices of size $k \times n$ (*couriers*) and a vector of size n (u). This type of representations is similar to the previous one, but instead of using a list of 2D matrices we have used a single 2D matrix with m extra lines and columns for the depot.

	1	2	3	4	D_1	D_2
1	0	0	0	0	0	1
2	0	0	1	0	0	0
3	0	0	0	0	1	0
4	1	0	0	0	0	0
D_1	0	1	0	0	0	0
D_2	0	0	0	1	0	0

Table 6: Example with two couriers carrying 4 items. In red: the travel of the first courier and in blue the travel of the second one.

	1	2	3	4
1	0	1	1	0
2	1	0	0	1

Table 7: Example with two couriers carrying 4 items. In red: items assigned to the first courier and in blue the ones assigned to the second one.

1	2	3	4
3	1	2	1

Table 8: One of the many possible correct u vectors. In red: items of the first courier and in blue the ones assigned to the second one.

As before, we need to be sure that:

- each item can only be delivered by one courier;
- each item must be delivered;

- each courier has to start form the depot and return there at the end of the delivery path;
- avoid sub-paths and loops.

5.2 Constraints

The first constraint ensures that each row/column has exactly one element equal to 1. We used it to ensure that each *Delivery Point* is reached by exactly one courier. Then, we integrated it with another constraint that forces the courier to leave the *Delivery Point* if it has previously reached it.

$$ExactlyOne(\forall j\ x_{i,j}) \quad \forall i = 1..(n+m) \tag{16}$$

$$ExactlyOne(\forall i\ x_{i,j}) \quad \forall j = 1..(n+m), \tag{17}$$

where $x_{i,j}$ is the i, j elements of the *table* matrix.
To represent the fact that a courier cannot loop in a *Delivery Point*, we set as false all the element in the diagonal.

$$x_{i,i} = False \quad \forall i = 1..(n+m) \tag{18}$$

Since all the couriers must leave the depot, we set as false all the elements that connect two depots.

$$x_{i,j} = False\ \ \forall i = n+1..(n+m)\ \ \forall j = n+1..(n+m);\ \ where\ \ i \neq j \tag{19}$$

	1	2	3	4	D_1	D_2
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
D_1	0	0	0	0	0	0
D_2	0	0	0	0	0	0

Table 9: In red all the elements set false with the (18) and (19) constraints.

We set exactly one element equal to 1 for each column of the couriers matrix:

$$ExactlyOne(\forall k\ c_{k,i}) \quad \forall i = 1..n, \tag{20}$$

where $c_{i,j}$ is the i, j element of the couriers matrix.
We have that different couriers start from different depot rows:

$$x_{i,j} \Rightarrow (c_{i-n,j} = True)\ \ \forall i = n+1..n+m\ \ \forall j = 1..n. \tag{21}$$

We have that if two items are connected, then they belong to the same courier:

$$x_{i,j} \Rightarrow (c_{k,i} = c_{k,j}) \quad \forall i, j = 1..n \ \ \forall k = 1..m \ \ . \tag{22}$$

We have the load constraint:

$$\sum_{i=1}^{n} c_{k,i} \cdot s_i \leq load_k \ \ \forall k = 1..m \tag{23}$$

where s_i is the weight of the i-items and $load_k$ is the maximum load of the k-courier.

We have finally the order constraint that avoid the presence of loops.

$$x_{i,j} \Rightarrow (u_i < u_j) \tag{24}$$

where u_i is the i-element of the u vector.

5.3 Objective Function

To get the objective function, we summed the distances traveled by all the couriers.

$$\text{OBJ} = \sum_{i=1}^{n+m} \sum_{j=1}^{n+m} x_{i,j} \cdot D_{i,j} \tag{25}$$

where again $x_{i,j}$ is the i, j element of the *table* matrix and $D_{i,j}$ is the distance between the i-item and the j-item.

6 Symmetry Breaking

In this section we point out the symmetry breaking constraint we tried to use in the models that are the following:

- Lexicographic ordering;
- Load ordering;
- Inverse path blocking;

No matter the effort we make and how much imagination we use, these constraints have no intention of improving our model. We think this is due to the complexity of the instances we have to solve. We are pretty sure that, especially for larger instances, finding an initial solution in less time dramatically improves model performance.

6.1 Lexicographic ordering

The first set of symmetry breaking constraint is about lexicographic constraints. We used them to reduce the search space by imposing an order on the items delivered. This means that, if we have two couriers with the same capacity, we force the first of them to have the list of transported items which is lexicographically greater than or equal to the list of the second.

Then, we also tried to impose lighter lexicographic constraints by considering only the rows representing the couriers. Of course, we have imposed those constraints between the couriers with the same capabilities in order to keep at least one optimal solution in the space.

Unfortunately, these constraints did not improved the performance of our models. This is probably due to the fact that since the instances are very large, it may be more important to find a starting solution in a few seconds than to go straight to the optimal one, which is too difficult to find.

6.2 Load ordering

The second set of symmetry breaking constraints concerns the load of the couriers. Like lexicographical ones, they reduce the search space by imposing an order on the items delivered. In this case we have forces each courier to deliver a total load greater than or equal to the couriers with the same or lower capacity.

This should reduce the search space in a significant way, however it is not enough to find the optimal solution and even worsened the performance.

6.3 Inverse path blocking

The last symmetry we tried to break is the inverse path, in fact a courier can simply follow the opposite path to get a new solution that is semantically equivalent to the previous one. To remove this behavior we compared each matrix we compare it to its inverse.

As before, this showed no signs of improvement.

7 Results

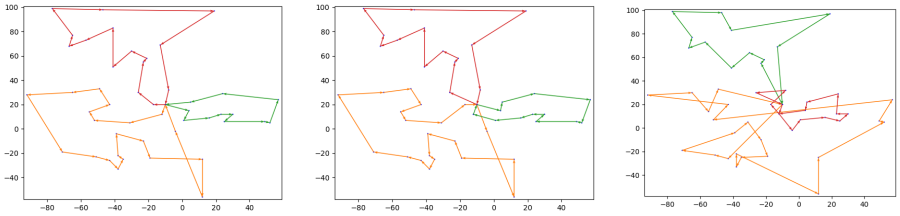
After implementing all the models, we have run all the given instances obtaining the following results:

	1	2	3	4	5	6	7	8	9	10	11
Or-Tools	1144	1910	3034	3640	3848	-	1440	2862	4092	-	1142
Gurobi	1152	2094	2906	3716	3803	-	1443	2994	4149	-	1162
Z3	1470	-	3598	-	-	-	2112	-	-	-	1554

Table 10: Summary of optimal path length found for all instances (the columns) by each model.

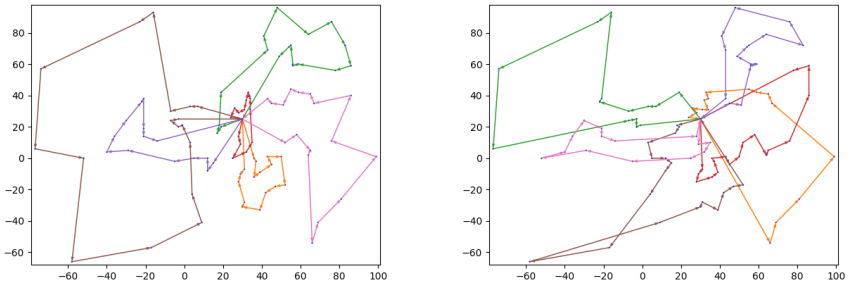
As we can observe from the summary table, the best results are obtained through the CP model because in Or-Tools API is present the global constraint circuit, where in the other models, it has been implemented by us. Anyway, despite the lack of this constraint, Gurobi results are very close to those obtained from OR-Tools. Regarding Z3, what we can say is that in cases where we find some solutions, these are far from the optimal one due to the low efficiency of the solver.

The images presented in the following pages are the graphical representations of the solutions. They are divided by instances and model as suggested by their descriptions. The textual result are located inside the *Result/Gurobi/Result.txt*, *Result/Or_Tools/Result.txt* and *Result/SMT/Result.txt*.



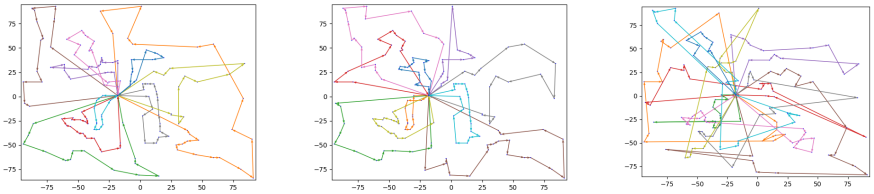
(a) Or-Tools (b) Gurobi (c) Z3

Figure 2: Instance 1



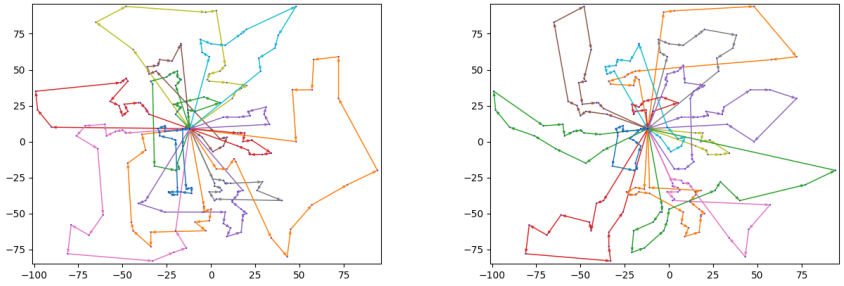
(a) Or-Tools (b) Gurobi

Figure 3: Instance 2



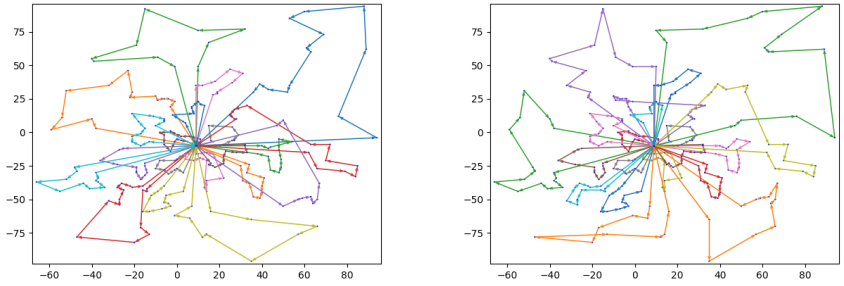
(a) Or-Tools (b) Gurobi (c) Z3

Figure 4: Instance 3



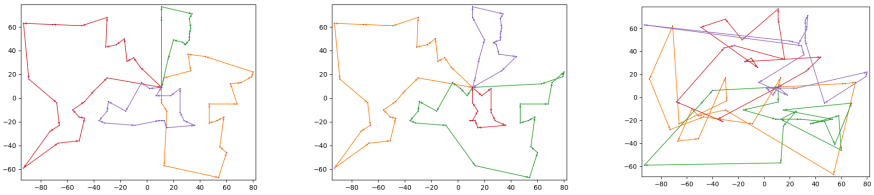
(a) Or-Tools (b) Gurobi

Figure 5: Instance 4



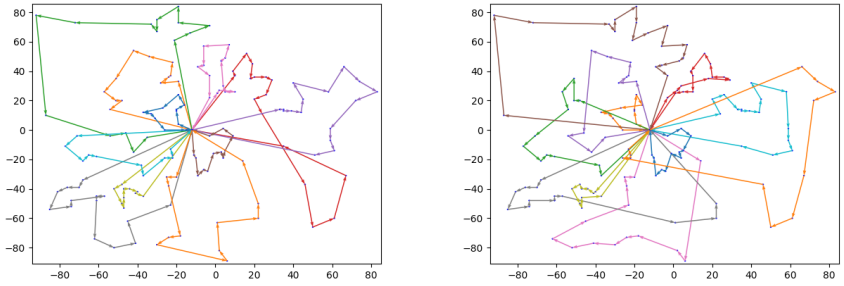
(a) Or-Tools (b) Gurobi

Figure 6: Instance 5



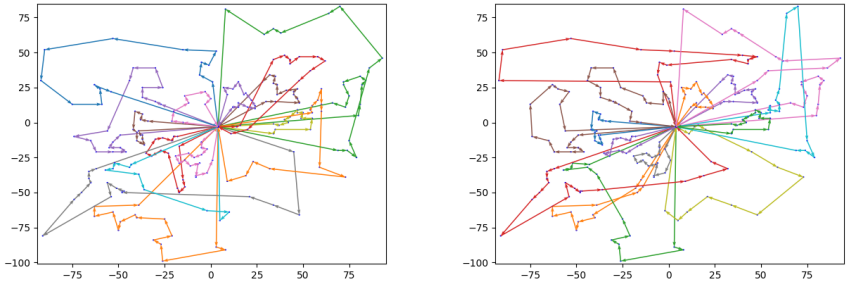
(a) Or-Tools (b) Gurobi (c) Z3

Figure 7: Instance 7



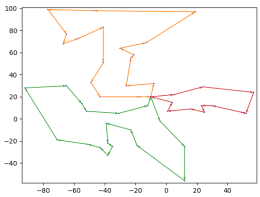
(a) Or-Tools (b) Gurobi

Figure 8: Instance 8

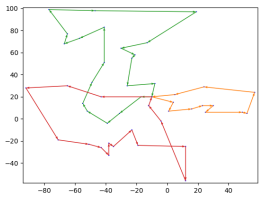


(a) Or-Tools (b) Gurobi

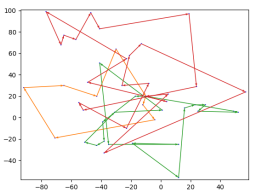
Figure 9: Instance 9



(a) Or-Tools



(b) Gurobi



(c) Z3

Figure 10: Instance 11

8 Future Developments

During the implementation of the SMT-model, we found a new way to represent the problem (using a single matrix) that allows us to save computational space and time. Indeed, with the old representation we had that the number of parameters to be optimized are

$$params = m \cdot (n+1)^2, \tag{26}$$

where m is the number of couriers and n is the number of items (the plus 1 is for representing the depot). With the new model instead, we have that

$$params = (n+m)^2 \tag{27}$$

As you can see, assuming a simple problem with $n = 10$ and $m = 3$, we got that

$$\begin{aligned} old &= 3 \cdot 11^2 = 363 \\ new &= (10+3)^2 = 169 \end{aligned} \tag{28}$$

and immediately come up that the saving is significant (more than 2 times and it grows as the size of the problem increases). So as first future development we thought to encode also Or-tools and Gurobi models in that way to increase performances.

Another interesting feature to add could be the reinjection of the solutions into a *pure constraint model*. In this way we could have both a good and quick initial solution, but retaining the possibility for the solver to exchange items between couriers.

9 Conclusions

In this paper we have reported our solution to the Multiple Couriers problem, using a top-to-bottom approach, where we first tried to better understand the problem itself by analyzing its critical points and applying some simplifications for the resolution. We have began the resolution by starting with the Constraint Programming model, then we have moved to Linear Programming and finally to SMT, by modifying the model created in the initial phase according to the needs of each solver (lack of constraints, solver limitations, etc.).

Results have shown that, OR-Tools, with its circuit constraint, manages to obtain, in almost all instances, a better result although the one recreated by us for Gurobi did its job very well. As for SMT, even though it had a much more simplified and performing model than the other two (without that model no instance had obtained a solution), it had the worst performances among the solvers. In fact, it found solutions in only four instances with disappointing results. The reason why SMT is much worse than the other models could be that it runs without multi-threading, so it uses a single CPU while the others use all the available cores.