CDMO project

Armando Renzullo, Cosimo Emanuele Russo,
Tommaso Perniola

July 2024

1 Introduction

Regarding the completion time, the project took three months with intermittent work. The general tasks related to the experiments with different solvers and strategies were always discussed together before each of us took responsibility for a specific task. However, any issues or challenges encountered were reviewed and discussed with the entire group, ensuring collective input and oversight. The construction of the model was a collaborative effort throughout.

The main difficulties were mostly related to solving the SAT instances, which required the help of everyone in the group and the models developed in MiniZinc because they served as the starting point for all future models.

The following paragraphs provide an overview of the encoding and solving procedure.

1.1 Semplifications and Preprocessing

To ensure the efficiency and feasibility of our models, we employed several simplification and preprocessing steps. This included checking mathematical properties such as the triangle inequality and matrix symmetry, as well as establishing lower and upper bounds for key variables. The following sections detail these steps.

1.2 Triangle Inequality and Symmetry

Using Python scripts, we verified that the triangle inequality held for all given distance matrices D. This property ensures that for any three points i, j and k, the direct distance from i to k is not greater than the distance from i to j plus the distance from j to k:

$$D[i,k] \leq D[i,j] + D[j,k] \tag{1}$$

Our scripts confirmed that the triangle inequality was always respected. Additionally, we checked for symmetry in the distance matrices and found that they were always non-symmetric, meaning

$$D[i,j] \neq D[j,i] \tag{2}$$

for some instances.

1.3 Distance and Load Bounds

To effectively distribute packages among couriers, we employed a straightforward yet highly effective heuristic. We introduced a constraint based on the number of packages (n) and the number of couriers (m). Specifically, we ensured that each courier would carry at least $\left\lfloor \frac{n}{m} \right\rfloor$ packages, with a maximum of $\left\lceil \frac{n}{m} \right\rceil$ packages per courier.

For determining the maximum and minimum distances each courier would need to travel, we conducted a specific preprocessing step tailored to our instances. This preprocessing revealed that the maximum distance between any two consecutive packages i and $i+1$ is always bounded by 200 units, while the minimum distance is bounded by 2 units. These bounds were consistently observed across all instances considered in our study.

2 CP Model

Constraint programming is an optimization paradigm that systematically explores all potential combinations of variable values through a backtracking tree search. Using the MiniZinc language, we developed a constraint programming model which was subsequently tested with two different solvers: Gecode and Chuffed. We experimented with various problem encodings, particularly comparing those that utilize symmetry-breaking constraints with those that do not.

2.1 Decision Variables

Our model relies on the following decision variables:

- $y[i]$: It is a 1D-array which represents the location assigned to position i in the sequence of paths, where $y[i] \in \{1, 2, \ldots, n+1\}$. Here, $n+1$ represents a special value, which indicates the depot. Moreover, $y[i]$ its domain is [1..n+m+1], such that:
  - $y[1]$: is the starting location for the first courier's path.
  - $y[n+m+1]$: is the ending location for all courier paths.
- $ind[i]$: It is a 1D-array with domain [0..m] which represents the indices that mark the boundaries between segments for each courier route, where $ind[i] \in \{1, 2, \ldots, m+n+1\}$. These indices are used to segment the path

sequence into parts for each courier, distinguishing between different parts of the overall route array.

- *max_distance_per_courier*$[i]$: It is a 1D-array which indicates the maximum distance allowed for courier i, where *max_distance_per_courier*$[i] \in \{n+1, n+2, \dots, \text{total}-n\}$. Notice that *total* is a constant that represents the sum of all elements in the matrix D for the indices i and j ranging from 1 to $n+1$. This means that this variable is constrained based on the distances traveled along the assigned paths.

- *max_distance*: represents the maximum distance traveled by any courier in the optimized route configuration. Its definition and usage are crucial for minimizing the overall distance while ensuring each courier's route adheres to specified constraints. It is defined as follows:

$$\text{max\_distance} = \max_i \textit{max\_distance\_per\_courier}[i]$$

2.2 Objective Function

The objective of the model is to minimize the maximum distance traveled by any courier. This can be formulated as the following optimization problem:

$$\min \quad \text{max\_distance}$$

In this formal mathematical representation: - The objective function is explicitly defined using the min operator to indicate minimization. - The variable to be minimized is *max_distance*, which represents the maximum distance traveled by any courier. - The objective function itself is constrained by the equality constraint defining *max_distance* as the maximum value among all *max_distance_per_courier*$[i]$ values.

2.3 Constraints

The constraint programming model includes the following constraints:

1. **Boundary conditions for courier paths**

$$y[1] = n+1, \quad y[n+m] = n+1$$

   Ensures that the starting and ending points of all courier paths are set to $n+1$, the depot/origin.

2. **Exclusion of end point from start and end positions**

$$y[2] \neq n+1, \quad y[n+m-1] \neq n+1$$

   Prevents the immediate successors of the start and the predecessors of the end from being $n+1$, ensuring valid path connections.

3. **Initial and final Indices of courier routes**

$$ind[0] = 1, \quad ind[m] = n + m + 1$$

Defines the starting and ending indices for segmenting courier routes.

4. **Depot marking in route segmentation**

$$\forall i \in \{1, \ldots, m-1\} : \quad y[ind[i]] = n + 1$$

Ensures that the depot $n+1$ marks the boundaries between segments for each courier route.

5. **Adjacent segment exclusion**

$$\forall i \in \{3, \ldots, n+m-1\} : \quad (y[i] = n+1 \rightarrow y[i-1] \neq n+1 \wedge y[i+1] \neq n+1)$$

Prohibits the depot $n+1$ from being adjacent to itself in the sequence, ensuring proper segmentation.

6. **Segment size and position**

$$\forall i \in \{0, \ldots, m-1\} : \quad ind[i] + \lceil \frac{n}{m} \rceil \leq ind[i+1] \quad \text{and} \quad ind[i+1] - ind[i] - 1 \leq n - m + 1$$

Controls the size and position of segments, ensuring they are within bounds and evenly distributed across the path sequence.

7. **Path segmentation exclusion**

$$\forall j \in \{1, \ldots, n+m+1\} \setminus \{ind[0], \ldots, ind[m-1]\} : \quad y[j] \neq n+1$$

Ensures that all positions not explicitly marked as segment boundaries do not contain the depot $n+1$.

8. **Distinct values**

$$all\_different\_except(y, \{n+1\})$$

Requires all elements in array y to be distinct, except for $n+1$ which can appear multiple times as the depot marker. This can be naturally achieved by using the global constraint *all_different_except*.

9. **Capacity is not exceeded**

$$\forall i \in \{1, \ldots, m\} : \quad \sum_{c \in \{ind[i-1], \ldots, ind[i]\} \setminus \{y[n+1]\}} s[y[c]] \leq l[i]$$

Ensures that the total capacity used by couriers in each segment does not exceed their respective limits $l[i]$.

10. **Distance calculation**

$$\forall i \in \{1, \ldots, m\} : \quad \sum_{c \in \{ind[i-1], \ldots, ind[i]-1\}} D[y[c], y[c+1]] = max\_distance\_per\_courier[i]$$

Calculates the distance traveled by each courier in their respective segments, ensuring it matches the maximum allowed distance.

Implied Constraints The model doesn't need implied constraints, since those that are already present actually models very well the problem and any further constraint doesn't help the search process.

Symmetry Breaking Constraints Our model does not allow symmetries, hence we didn't include any symmetry breaking constraints.

2.4 Validation

The model is implemented in MiniZinc and it has been tested using two different solvers, Gecode and Chuffed. In addition, it has been made a comparison between different search strategies over the integer space.

Experimental Design The most effective search strategy turned out to be the one which implements:

- On y: "domWdeg" as variable selection heuristic with "indomain_random" as value selection heuristic.
- On *ind*: "first fail" as variable selection heuristic with "indomain_random" as value selection heuristic.
- a "restart_luby" as restart strategy, in order to periodically restart the search.
- a "relax and reconstruct" heuristic over y to improve the efficiency of finding solutions by partially relaxing the current assignment and then reconstructing it.

We implemented a restart strategy and a "relax and reconstruct" heuristic in order to explore different regions of the search space, helping to escape from local optima and improve the search diversity.

Experimental Results The following table shows the results for two different solvers, each tested with different search strategies on y.

3 SMT Model

The SMT (Satisfiability Modulo Theories) approach solves optimization problems by employing a suite of solvers, each specialized in different theories. In addressing the couriers problem, the models leveraged the linear integer theory and the array theory. These theories were instrumental in minimizing the spatial complexity of the models, allowing for the use of more advanced structures and constraints during the modeling process.

Inst.	**Gecode 1**	**Gecode 2**	**Chuffed 1**	**Chuffed 2**
1	14	14	14	14
2	226	226	226	226
3	12	12	12	12
4	220	220	220	220
5	206	206	206	206
6	322	322	322	322
7	167	167	167	167
8	186	186	186	186
9	UNKNOWN	436	436	436
10	244	244	244	244
11	305	UNKNOWN	621	UNKNOWN
12	346	UNKNOWN	404	UNKNOWN
13	454	476	1110	1204
14	434	UNKNOWN	1000	UNKNOWN
15	481	UNKNOWN	988	UNKNOWN
16	286	UNKNOWN	286	UNKNOWN
17	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
18	386	UNKNOWN	829	UNKNOWN
19	334	UNKNOWN	334	UNKNOWN
20	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
21	374	UNKNOWN	693	UNKNOWN

Table 1: Results using Gecode and Chuffed with different search strategies on y

3.1 Decision Variables

Decision variables are the variables that represent choices or decisions to be made to solve the problem. These variables determine the structure of the solution and directly impact the objective function and constraints.

The variable *b_path* indicates if a courier takes a specific item, such that *b_path[c][j]* = *True* if the courier c takes the *j-th* item. It is a boolean matrix that has 'IntSort()' indices and each element is an 'ArraySort()' itself. This 'ArraySort(IntSort(), BoolSort())' has integer indices indices and boolean elements. As it can be easily seen, the theory used are the Array theory and the Boolean Logic theory.

The variable *path* instead is a two-dimensional array of integers that represents the sequence of the items taken by each courier. In particular, *path[c][j]* indicates the *j-th* item taken by the courier c It has integers indices and each element is a nested 'ArraySort(IntSort(), IntSort())', that defines the type of the elements in the array. The theories used here are the Arrays and Arithmetic theories.

Then, *load*, *size*, *path lenght* and *total-distance* are all 1-dimensional arrays of integers such that they all uses the Arrays theory and the Arithmetic theory. In particular:

- *load*: represents the load capacity of each courier, such that *load[c]* gives the load capacity of the courier c.
- *size*: represents the size of each item, such that *size[j]* gives the size of the item j.
- *path lenght*: indicates the length of the path for each courier, such that *path_lenght[c]* gives the length of the path for the courier c.
- *total distance*: indicates the total distance traveled by each courier, i.e. *total_distance[c]* is the total distance for the courier c.

Moreover, the variable D_{func} uses the 'Function()' sort to represent the distance matrix, which maps each pair of items to an integer distance. This function exists primarily for convenience, as it helps in mapping Z3 arrays to Python arrays, addressing the issues arising from Python's 0-indexing. This function takes two integers, the two items, and outputs another integer, the distance, such that $D_{\text{func}}(i, j)$ gives the distance between the item i and the item j. The theory used here is the EUF (Uninterpreted Functions) theory.

Finally, the last decision variable *max_dist* represents the maximum distance traveled by any courier, and it is a single integer representing the objective function to be minimized. It uses the 'Int' sort and the Arithmetic theory.

3.2 Objective Function

As before, the goal is to assign items to couriers such that the maximum distance traveled by any courier is minimized. In a more formal notation:

$$\min_{c \in \text{Couriers}} \max(D_c)$$

where D_c represents the total distance traveled by courier c. Again, this objective function aims to find a solution where the longest distance traveled by any courier is minimized. In particular, the total distance traveled by courier c, denoted as D_c, can be formalized as:

$$D_c = \sum_{j=1}^{k_c} D_{\text{func}}(p_{c,j-1}, p_{c,j})$$

Here:

- k_c is the number of nodes visited (the picked-up items) by courier c.
- $p_{c,j}$ represents the j-th location visited by courier c.
- $D_{\text{func}}(i, j)$ is the distance function that calculates the distance between locations i and j.

This formula calculates the total distance traveled by courier c based on its assigned route.

3.3 Constraints

Constraints ensure the solution is valid and respects the problem's requirements. First, we have to list the constraints defined just for convenience, to address the 0-indexing in Python:

- **Distance matrix mapping:** $\forall i, j \in \{0, 1, \ldots, n\} \quad D_{\text{func}}(i, j) = D[i][j]$
- **Items' sizes:** $\forall i \in \{0, 1, \ldots, n\} \quad \text{size}[i+1] = s[i]$
- **Couriers' load capacities:** $\forall c \in \{0, 1, \ldots, m\} \quad \text{load}[c+1] = l[c]$

Now we give a mathematical formulation of those constraints that actually models the problem:

Path range constraints: The path values for each courier must range between 0 and $n + 1$:

$$0 \leq \text{path}[c][j] \leq n + 1 \quad \forall c \in \text{Couriers}, \forall j \in \{1, \ldots, \text{MAX\_ITEMS}\} \tag{3}$$

Path length boundaries: The length of each courier's path must be between 3 (because we have to consider the origin node twice) and the maximum number of items:

$$3 \leq \text{path\_length}[c] \leq \text{MAX\_ITEMS} \quad \forall c \in \text{Couriers} \tag{4}$$

Initial and final node constraints: Each courier's path must start and end at the depot (node $n+1$):

$$\text{path}[c][1] = n+1 \quad \text{and} \quad \text{path}[c][\text{path\_length}[c]] = n+1 \quad \forall c \in \text{Couriers} \tag{5}$$

Unvisited items set to zero: Any positions in the path beyond the path length are set to zero:

$$i > \text{path\_length}[c] \implies \text{path}[c][i] = 0 \quad \forall c \in \text{Couriers}, \forall i \in \{1, \dots, \text{MAX\_ITEMS}\} \tag{6}$$

Load capacity constraint: The total load for each courier must not exceed its capacity:

$$\sum_{j \in \text{Items}} \text{If}(b\_path[c][j], \text{size}[j], 0) \leq \text{load}[c] \quad \forall c \in \text{Couriers} \tag{7}$$

Unique item assignment. Each item must be assigned to exactly one courier:

$$\sum_{c \in \text{Couriers}} \text{If}(b\_path[c][j], 1, 0) = 1 \quad \forall j \in \text{Items} \tag{8}$$

Maximum items constraint: No courier can carry more than the maximum number of items:

$$\sum_{j \in \text{Items}} \text{If}(b\_path[c][j], 1, 0) \leq \text{MAX\_ITEMS} \quad \forall c \in \text{Couriers} \tag{9}$$

No revisit constraint: Each courier cannot visit the same node more than once:

$$\text{distinct\_except}(\{\text{path}[c][j] : 1 \leq j \leq \text{MAX\_ITEMS}\}, \{0\}) \quad \forall c \in \text{Couriers} \tag{10}$$

Channeling constraints: Ensures consistency between the boolean assignment and the actual path taken by each courier c, such that the *i-th* item is in the path if and only if the corresponding *b_path[c][i]* is evaluated to *True*.

$$b\_path[c][i] \implies \left(\bigvee_{j=1}^{\text{MAX\_ITEMS}} \text{path}[c][j] = i \right) \quad \forall c \in \text{Couriers}, \forall i \in \text{Items} \tag{11}$$

$$\neg b\_path[c][i] \implies \left(\bigwedge_{j=1}^{\text{MAX\_ITEMS}} \text{path}[c][j] \neq i \right) \quad \forall c \in \text{Couriers}, \forall i \in \text{Items} \tag{12}$$

Courier load not exceeded Ensures the load does not exceed the courier's capacity, taking into account only the visited nodes:

$$\sum_{j \in \text{Items}} \text{If}(b\_path[c][j], \text{size}[j], 0) \leq \text{load}[c] \quad \forall c \in \text{Couriers} \tag{13}$$

$$\sum_{j=2}^{\text{MAX\_ITEMS}} \text{If}(j < \text{path\_length}[c] - 1, \text{size}[\text{path}[c][j]], 0) \leq \text{load}[c] \quad \forall c \in \text{Couriers} \tag{14}$$

Distance computation: Calculates the total distance for each courier's path, including the additional distance if the path includes zeros between non-zero items:

$$\begin{aligned} \text{total\_distance}[c] = & \sum_{j=1}^{\text{MAX\_ITEMS}-1} \text{If}(\text{path}[c][j] \neq 0 \wedge \text{path}[c][j+1] \neq 0, \\ & D(\text{path}[c][j] - 1, \text{path}[c][j+1] - 1), 0) \\ & + \sum_{j=1}^{\text{MAX\_ITEMS}} \text{If}(\text{path}[c][j] = 0 \wedge \text{path}[c][j-1] \neq 0 \wedge \text{path}[c][j+1] \neq 0, \\ & D(\text{path}[c][j-1] - 1, \text{path}[c][j+1] - 1), 0) \quad \forall c \in \text{Couriers} \end{aligned} \tag{15}$$

Symmetry Breaking: Prevents symmetric solutions by ordering the assignment of items to couriers with equal load capacities:

$$\begin{aligned} & \forall c_1, c_2 \in \text{Couriers}, c_1 < c_2 \\ & \quad \text{load}[c_1] = \text{load}[c_2] \implies \text{lexleq}([b\_path[c_1][j] : j \in \text{Items}], [b\_path[c_2][j] : j \in \text{Items}]) \end{aligned} \tag{16}$$

3.4 Validation

Experimental Design The SMT model was initially constructed using Python's Z3 library and its optimizer. However, without converting it to SMT-LIB format initially, we faced challenges when scaling up to larger instances and exploring the entire search space. To address this, we implemented an **Iterative Deepening** search approach using the Z3 solver. The progressive tightening of the maximum distance mimics the deepening aspect of iterative deepening search, ensuring that the search space is explored systematically and exhaustively. The search tries to find improved solutions within a specified time limit, until the model's constraints rendered it *unsatisfiable.*

Experimental Results For the first 10 instances, we obtained the same results for both the model that uses only the Z3 optimizer and the one that iterates over the search space. These results are displayed in Table 2.

4 MIP Model

In this study, we implemented a mixed-integer programming (MIP) model using the Gurobi solver to optimize courier routes.
Gurobi is a state-of-the-art solver for mathematical programming. It is widely

Instance	**Z3 solver+SB**	**Z3 solver w/out SB**
1	**14**	**14**
2	**226**	**226**
3	**12**	**12**
4	**220**	**220**
5	**206**	**206**
6	**322**	**322**
7	**167**	**167**
8	**186**	**186**
9	**436**	**436**
10	**244**	**244**
11	**923**	1019
12	541	**489**
13	**1226**	1256
14	N/A	N/A
15	N/A	N/A
16	**286**	**286**
17	N/A	N/A
18	N/A	N/A
19	**429**	437
20	N/A	N/A
21	N/A	**1264**

Table 2: Results of the Z3 solver with Iterative Deepening and a timeout of 300 seconds, with and without Symmetry Breaking constraints

used in academia and industry due to its robustness and efficiency in solving large-scale optimization problems. Gurobi employs several advanced techniques to find optimal solutions, including the simplex method and the dual simplex method.
The dual simplex method is a variation of the simplex method that solves linear programs by maintaining feasibility with respect to the dual problem. Instead of iterating over primal variables, it iterates over dual variables. This method is particularly useful when the initial solution is not feasible for the primal problem but is feasible for the dual. It allows Gurobi to efficiently handle problems that are modified incrementally, such as adding or relaxing constraints.
For mixed-integer programming (MIP) problems, Gurobi uses a branch-and-bound algorithm combined with cutting planes, known as branch-and-cut. The branch-and-bound algorithm systematically explores branches of the solution space, solving linear relaxations of the problem to obtain bounds on the optimal solution. When the solution to a relaxation is not integer, the problem is divided into smaller subproblems (branching). Cutting planes are additional constraints added to tighten the linear relaxation and exclude infeasible regions.

This section describes the model's decision variables, objective function, and constraints in detail.

4.1 Decision Variables

The primary decision variables used in the model are:

- **Path Selection Variables** x_{cij}:
  - Definition: x_{cij} is a binary 3D matrix where $x_{cij} = 1$ if the path from node i to node j is selected in the optimal solution by courier c, and $x_{cij} = 0$ otherwise.
  - Dimension: $m \times n{+}1 \times n{+}1$, where m is the total number of couriers, n is the total number of nodes in the network, and $+1$ is given by the adding of the depot node.
  - Purpose: These variables construct the actual tour by indicating the paths taken between nodes by each courier.
  - Mathematical Representation:

$$x_{cij} \in \{0, 1\}, \quad \forall i, j \in \{1, 2, \ldots, n + 1\}, \quad \forall c \in \{1, 2, \ldots, m$$

- **Cumulative Distance Auxiliary Variable** $u_c i$:
  - Definition: $u_c i$ is an integer variable where $u_c i = D[prev][i]$ if the courier c travels the path going from node $prev$ to node i, and $u_c i = 0$ otherwise.
  - Dimension: $m \times n + 1$, where m is the total number of couriers, and $n + 1$ is the total number of nodes plus the depot.

- Purpose: These variables keeps track of the distances involved in the overall path of the couriers, and their sum indicates the total path length of courier c.
- Mathematical Representation:

$$u_c i \in \{0, \ldots, max_d istance\}, \quad \forall c \in \{1, 2, \ldots, m\}, \quad \forall i \in \{1, 2, \ldots, n{+}1\}$$

- **Cumulative Load Auxiliary Variable** $v_c i$:
  - Definition: $v_c i$ is an variable where $v_c i = s[i]$ if the courier c visits node i, and $v_c i = 0$ otherwise.
  - Dimension: $m \times n$, where m is the total number of couriers, and n is the total number of nodes.
  - Purpose: These variables keeps track of the total load carried by the courier c.
  - Mathematical Representation:

$$v_c i \in \{0, \ldots, max_l oad\}, \quad \forall c \in \{1, 2, \ldots, m\}, \quad \forall i \in \{1, 2, \ldots, n\}$$

- **Objective Variable** z:
  - Definition: z is a continuous variable.
  - Purpose: This variables keeps track of the maximum distance traveled by all the couriers.

In the context of the parsed instance, these variables will adapt to the specific nodes and distances provided.

4.2 Objective Function

The objective function aims to minimize the maximum distance traveled. Mathematically, it is represented as:

$$\text{Minimize} \quad \max_{c \in \{1, \ldots, m\}} \sum_{i=1}^{n+1} \sum_{j=1}^{n+1} d_{ij} \cdot x_{cij}$$

where:

- d_{ij} is the distance between node i and node j.

4.3 Constraints

- **Objective Variable Update**:

$$\sum_{j=1}^{n+1} u_{cj} \leq z \quad \forall c \in \{1, \ldots, m\}$$

- **Every node must be visited exactly once (except for the depot n+1)**:

$$\sum_{c=1}^{m}\sum_{i=1}^{n} x_{cij} \leq 1 \quad \forall j \in \{1,\dots,n\}$$

$$\sum_{c=1}^{m}\sum_{i=1}^{n} x_{cij} \geq 1 \quad \forall j \in \{1,\dots,n\}$$

- **Every courier must visit the depot as the last node**:

$$\sum_{i=1}^{n} x_{ci,n+1} = 1 \quad \forall c \in \{1,\dots,m\}$$

- **Each courier must start from the depot**:

$$\sum_{j=1}^{n} x_{c,n+1,j} = 1 \quad \forall c \in \{1,\dots,m\}$$

- **Couriers cannot stay still**:

$$x_{cii} \leq 0.5 \quad \forall c \in \{1,\dots,m\}, \forall i \in \{1,\dots,n+1\}$$

- **Cumulative Distance Update**:

$$(x_{cij} = 1) \Rightarrow (u_{cj} = d_{ij}) \quad \forall c \in \{1,\dots,m\}, \forall i,j \in \{1,2,\dots,n+1\}$$

- **Path Contiguity**:

$$(x_{cij} = 1) \Rightarrow \left(\sum_{k=1}^{n+1} x_{cki} \geq 1\right) \quad \forall c \in \{1,\dots,m\}, \forall i,j \in \{1,2,\dots,n+1\}$$

$$(x_{cij} = 1) \Rightarrow \left(\sum_{k=1}^{n+1} x_{cjk} \geq 1\right) \quad \forall c \in \{1,\dots,m\}, \forall i,j \in \{1,2,\dots,n+1\}$$

  This is like a forward and backward check in order to further speed up the search.

- **Load Update**:

  $(x_{cij} = 1) \Rightarrow (v_{cj} = s_j) \quad \forall c \in \{1,\dots,m\}, \forall i,j \in \{1,2,\dots,n{+}1\}$ where $dest \leq n$

- **Speed up Search Constraint**:

$$\sum_{j=1}^{n+1} x_{cij} \leq 1 \quad \forall c \in \{1,\dots,m\}, \forall i \in \{1,2,\dots,n+1\}$$

  Just to delete useless assignments from the search space.

Instance	GurobiPy+SB	GurobyPy w/out SB
1	**14**	**14**
2	**226**	**226**
3	**12**	**12**
4	**220**	**220**
5	**206**	**206**
6	**322**	**322**
7	**167**	**167**
8	**186**	**186**
9	**436**	**436**
10	**244**	**244**
11	N/A	N/A
12	541	N/A
13	**638**	**572**
14	N/A	N/A
15	N/A	N/A
16	N/A	**286**
17	N/A	N/A
18	N/A	N/A
19	N/A	N/A
20	N/A	N/A
21	N/A	**1264**

- **Loop Avoidance**:

$$\sum_{k=1}^{i+1} x_{cjk} + x_{cij} \leq 1 \quad \forall c \in \{1, \dots, m\}, \forall i \in \{1, 2, \dots, n\}, \forall j \in \{1, 2, \dots, i\}$$

- **Symmetry Breaking**:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} x_{cij} \geq \sum_{i=1}^{n}\sum_{j=1}^{n} x_{c+1,ij} \quad \forall c \in \{1, \dots, m-1\}$$

  Under the assumption that the couriers (i.e. channels of the matrix) are disposed in decreasing order of load size.

- **Maximum Load**:

$$\sum_{j=1}^{n} v_{cj} \leq l_c \quad \forall c \in \{1, \dots, m\}$$

4.4 Validation

We can see

5 Conclusions

In conclusion, our project successfully utilized Constraint Programming (CP) with MiniZinc, Satisfiability Modulo Theories (SMT) using Z3, and Mixed-Integer Programming (MIP) with Gurobi to optimize routing for multiple couriers across 21 instances. Notably, our results were promising, demonstrating effective solutions across diverse scenarios. Surprisingly, we achieved improved outcomes without relying on symmetry breaking constraints, highlighting the robustness and efficiency of our approach. This project underscores the versatility and effectiveness of combining different optimization techniques to tackle complex logistics and routing challenges.