CDMO Project Report

Alessandro Folloni alessandro.folloni2@studio.unibo.it
Daniele Napolitano daniele.napolitano4@studio.unibo.it
Leonardo Petrilli leonardo.petrilli@studio.unibo.it
Pelinsu Acar pelinsu.acar@studio.unibo.it

February 1, 2024

1 Introduction

The MCP aims to assign a set of items to a set of couriers, and to determine the optimal routes for each courier, such that the total overall distance covered by the couriers.
We implemented four models to solve the same problem, for the following approaches: Constraint Programming (CP), Boolean Satisfiability (SAT), Satisfiability Modulo Theories (SMT) and Mixed-Integer Programming (MIP).

1.1 Input

The standard input for this problem are the values: *num_items, num_couriers, load_size, item_size* and *distances*. We also defined this common notation for all the models:

- **NODES** is the range of available location points including the depot.
- **CUSTOMER** is the range of available customer location points.
- **VEHICLE** is the range of available couriers.

1.2 Approach

For all the models, we based our choices on a paper about the Vehicle Routing Problem [1]. It proposed two different approaches:

1. The first is based on the idea of a successor matrix, that links each location to the next one visited, for each courier.
2. The second instead focuses on a predecessor array, that links each visited location with the previous one (without considering the courier).

Following an initial understanding and formulation of the problem, we tried two different models using Constraint Programming, and saw that the second one was better. So, we evaluated and compared both methods in CP and left out the first one for the other approaches.
The entire project took approximately eight weeks, during which each team member actively contributed to different solution approaches. After completing the implementation of all four models, the validation and writing of the report were split for each different method between the group members.
The primary challenge was translating these constraints into the corresponding solution approaches and making the model more efficient by trying different search methodologies and techniques.

2 CP Model

In this section, we present only the model B but the details of model A can be found in Appendix. As it is discussed in the introduction, both models have common input parameters.

2.1 Decision Variables

The decision variables of these two models are completely different except the one below for the subtour constraint:

- **num_visit** $\in$ **CUSTOMER** is a decision variable array of size **CUSTOMER**, getting a value for each node, except for the depot.

Model A:

- **succ** is a matrix of dimension **VEHICLE** x **NODES**, representing the route of each vehicle (See Appendix A).

Model B:

- **pred** $\in$ **NODES** is a decision variable array of size **CUSTOMER**, representing the predecessor of each customer meaning that pred[i]=j iff the vehicle which visits j, visits i immediately afterwards.

- **last** is a boolean decision variable array of size **CUSTOMER**, representing whether this customer is the last on the route meaning that last[i]=1 iff the visiting vehicle returns the depot immediately after visiting customer i and 0 otherwise.

- **is_pred** is a boolean decision variable array of size **CUSTOMER**, representing the inverse of **last**.

- **vehicle** $\in$ **VEHICLE** is a decision variable array of size **CUSTOMER**, with the meaning that vehicle[i]=j iff vehicle i has visited customer j.

- **used_couriers** $\in$ **VEHICLE** is a decision variable representing the number of couriers used.

Table 1: pred, last, is_pred and vehicle decision variable arrays of Model B for Instance-3

Customer	1	2	3	4	5	6	7
pred	8	8	8	2	4	3	1
last	0	0	0	0	1	1	1
is_pred	1	1	1	1	0	0	0
vehicle	3	1	2	1	1	2	3

2.2 Objective Function

The objective of both models is to minimize the maximum distance traveled by any courier. This can be formalized for model B as follows:

1. The total distance traveled by each courier can be calculated by summing up the distance that belongs to the same courier in pred to the index of the element; that is, from the predecessor to the current customer:

$$\text{dist1\_per\_courier[j]} = \sum_{i \in \textbf{CUSTOMER} \; where \; vehicle[i]=j} distances[pred[i], i]$$

2. Moreover, we need to add the distance from the current customer back to the depot iff the current customer is the last one on the route meaning that last[i] = 1:

$$\text{dist2\_per\_courier[j]} = \sum_{i \in \textbf{CUSTOMER} \; where \; vehicle[i]=j} distances[i, num\_items{+}1] * last[i]$$

3. Finally, we can find the maximum distance traveled by any courier by summing these two values:

$$\text{max\_dist} = \max(\text{dist1\_per\_courier[j]} + \text{dist2\_per\_courier[j]}) \text{ where j in } \textbf{used\_couriers}$$

The objective function is to minimize **max_dist** to achieve a fair division among the couriers.

2.3 Constraints

Constraints used in predecessor-based methodology (Model B) are defined as follows:

1. Since our decision variable **last** indicates the last customers on their routes, the sum of this variable for all customers must be equal to the number of used couriers. We impose this via the constraint below:

$$\text{used\_couriers} = \sum_{i \in \textbf{CUSTOMER}} last[i]$$

2. Another constraint of the problem is that if a customer i is a predecessor of another customer meaning that if **is_pred**[i] = 1, **i** has to occur in **pred** exactly once.

Considering table 1, since customer 2 is a predecessor of another customer (**is_pred**[2] = 1), customer 2 has to occur in **pred** array and we can observe that **pred**[4] = 2 ensures this.

Furthermore, each used vehicle must leave the depot. Since **num_items+1** represents the location of the depot in **pred** array, there must be exactly **used_couriers** entries in **pred** with the value **num_items+1**

We can implement these two constraints at the same time using the global cardinality constraint below:

global_cardinality(pred, NODES, is_pred ++ [used_couriers]);

3. To make sure that each subtour has been done by only one car and to check the capacity of each courier in another constraint, we need to match the customers with the couriers correctly.

One way of doing this is to check the **pred** array. For each customer i, if a vehicle visits customer i, then it must have visited its predecessor before as well. Therefore, customer i and the predecessor of customer i must be equal to each other in the **vehicle** array.

The following constraint ensures this true mapping:

$$\forall i \in \textbf{CUSTOMER} : \ vehicle[i] = vehicle[pred[i]] \ if \ pred[i] \neq num\_items + 1$$

4. Since **is_pred** and **last** are boolean decision variable arrays and the inverse of each other by definition, their sum must be equal to one for each customer i:

$$\forall i \in \textbf{CUSTOMER} : \ is\_pred[i] + last[i] = 1$$

5. To eliminate the subtours, we implement Miller-Tucker-Zemlin formulation [2], which uses an extra variable **num_visit**. If a vehicle drives from node i to node j, the value of **num_visit[j]** has to be bigger than the value of **num_visit[i]**. So each time a new node is being visited, the value for **num_visit[i]** increases.

This formulation would make it impossible for every value of to be larger than the previous one and it ensures that a vehicle will not drive in a circle. Since the depot does not get a value of **num_visit[i]**, it is possible to drive in a circle if the vehicle starts and ends at the depot.

$$\forall i \in \textbf{CUSTOMER} : \ num\_visit[i] > num\_visit[pred[i]] \ if \ pred[i] \neq num\_items+1$$

6. We need a capacity constraint to ensure that the total load for each courier does not exceed its vehicle capacity. We impose this by summing up the item sizes that belong to the same courier and constraint this number to be less than the load size of that vehicle:

$$\forall j \in \textbf{used\_couriers} : \ (\sum_{i \in \textbf{CUSTOMER} \ where \ vehicle[i]=j} item\_size[i]) < load\_size[j]$$

Redundant constraint

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.

Since the predecessor of a customer cannot equal the customer itself, we can impose this by writing a constraint. However, our model has already ensured this with the subtour constraint. Therefore, this constraint becomes redundant for our model but since it improves the performance of our model by helping the solver to eliminate inconsistent results more quickly, we decide to implement as follows:

$$\forall i \in \textbf{CUSTOMER} : \ pred[i] \neq i$$

Symmetry breaking constraint

Since the instances of our model do not guarantee equal capacities for each courier, the assignments between paths and couriers make a difference. However, some couriers may have equal capacities, and in that case, interchanging the paths that are assigned to these couriers to generate different solutions would cause an additional workload on our model. To eliminate these identical solutions, we define a constraint as below:

$$\forall i, j \in \textbf{VEHICLE} \ where \ i < j : \ if \ load\_size[i] = load\_size[j] : \\ lex\_less([vehicle[k] = i | k \in \textbf{CUSTOMER}], [vehicle[k] = j | k \in \textbf{CUSTOMER}]) \tag{1}$$

This constraint enforces some of the variables in **VEHICLE** to be lexicographically less than each other if the load sizes of those vehicles are equal to each other meaning that for any variable i, j in **VEHICLE** where load_size[i] = load_size[j] if we swap the values of i and j in **VEHICLE** array, solutions are preserved.

2.4 Validation

To evaluate the performances of our models, we implemented them in Minizinc and ran using both Gecode and Chuffed. While assessing the performance, different search strategies and restart techniques with linear and Luby sequences were practiced. The details of the experimental setup for each model are below:

Model A:

- int_search + restart_linear with scale of (num_items)*(num_items) (See Appendix B)

Model B:

- int_search + restart_linear with scale of (num_items)*(num_items), without symmetry breaking constraint

- int_search + restart_linear with scale of (num_items)*(num_items), with symmetry breaking constraint

Table 2: Results for Model B using Gecode and Chuffed with and without symmetry breaking using int search strategy

ID	Chuffed + SB	Chuffed w/out SB	Gecode + SB	Gecode w/out SB
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	406	422	167	167
8	**186**	**186**	186	186
9	436	461	436	436
10	244	244	244	244
13	1478	2156	486	444
16	927	668	529	377
19	N/A	N/A	N/A	785

As it can be seen from tables 2 and B.1, even though both models have the same objective value on smaller instances, model A performs faster and can reach the optimality. However, model B can go further on the large instances by offering a wider range of solving capabilities but still fails to reach optimality. Moreover, considering the variety of results for Chuffed and Gecode, Chuffed is better at solving small instances in less than 300 seconds. Gecode could be a better option as instances scale in size and complexity.

3 SAT Model

The SAT model follows the same approach of the CP model, adapting decision variables to be only boolean, by using either:

- **One-hot encoding**: a way of representing an integer variable that can take values from 0 to n as a binary vector of length n, where only the element at the position of the value is 1 and the rest are 0.
- **Base 2 encoding**: a way of representing decimal numbers with only two digits (0 and 1), where each bit represents a power of 2. The number of bits needed to encode an integer number n is equal to the ceiling of $log_2(n)$, which is the smallest integer that is $\geq log_2(n)$.

3.1 Decision Variables

- **pred** is the predecessor matrix: a $(n+1) \times n$ matrix of one-hot encoded boolean variables, where n is the number of items and $n+1$ is the number

of locations (including the depot). Each entry $pred_{i,j}$ indicates whether item j is preceded by location i in the tour.

- **courier** is a $m \times n$ matrix of one-hot encoded boolean variables, where m is the number of couriers and n is the number of items. Each entry $veichle_{i,j}$ indicates whether item j is assigned to courier i.
- **num_visit_bool** is a vector of length $\lceil log_2(n) \rceil$ of boolean variables. Rows represent items, and columns are the **base 2 encoding** of the order of visit. For example, if the decimal transformation of $num\_visit\_bool_0$ is 1, and the one of $num\_visit\_bool_1$ is 0, then it means that location 0 is visited before the location 1.

3.2 Constraints

1. Each item must be assigned to only one courier, i.e. for each column j of the **courier** matrix, exactly one element in the row must be true.

$$exactly\_one(\{\forall\ i \in \textbf{COURIERS} :\ courier_{i,j}\}) \qquad for\ j = 0, ..., n$$

2. Each courier must carry a total item size less or equal than its load size:

$$\sum_{j=0}^{n} courier_{i,j} \cdot item\_size_i \leq load\_size_i \qquad for\ i = 0, ..., m$$

3. for the **pred** matrix:
   (a) we enforce the one-hot encoding in each column by checking if for each column, exactly one element is true in a row:

$$exactly\_one(\{\forall\ i \in \textbf{NODES} :\ pred_{i,j}\}) \qquad for\ j = 0, ..., n$$

   (b) Since an item can't be the predecessor of itself, the main diagonal is set to false

$$\neg pred_{i,j}\ if\ i = j \qquad for\ j = 0, ..., n \qquad for\ i = 0, ..., n$$

   (c) Except for the depot's (last) row, in each one there must be at most one true value (can't have same predecessor, but can be without predecessor in the case $last_i = true$)

$$at\_most\_one(\{\forall\ j \in \textbf{ITEMS} :\ pred_{i,j}\}) \qquad for\ i = 0, ..., n$$

4. Link **pred** matrix with **couriers**: items carried by the same courier must be linked also in the pred matrix.

$$(pred_{i,j} \wedge courier_{k,j}) \implies courier_{k,i} == courier_{k,j}$$

$$for\ j \in \textbf{ITEMS},\ for\ i \in \textbf{ITEMS},\ for\ k \in \textbf{COURIERS}$$

5. in the **last** vector, if an item i has not a predecessor, then $last_i = true$. This constraint is enforced by checking if the row i of the **pred** matrix has only false values: this means that item i has no predecessor.

$$last_i = \begin{cases} True & if \bigwedge_{j \in \textbf{ITEMS}} (pred_{i,j} == False) \\ False & otherwise \end{cases}$$
$$for\ i = 0, ..., n$$

6. Among the items carried by a courier, exactly one of them must have *last* set to true (i.e. there must be only one last item in a route).

$$exactly\_one \left(\forall j \in \textbf{ITEMS} : \begin{cases} last_j & if\ courier_{i,j} == True \\ False & otherwise \end{cases} \right)$$
$$for\ i = 0, ..., n$$

7. MTZ formulation[2]to remove subtours. We also need to convert back the row values of **num_visit_bool** to integer before performing any comparison.

$$pred_{i,j} \implies bool\_to\_int(\{\forall k \in range(n\_int\_encoding) :\ num\_visit\_bool_{j,k}\},$$
$$\{\forall k \in range(n\_int\_encoding) :\ num\_visit\_bool_{i,k}\},\ n\_int\_encoding)$$
$$for\ j = 0, ..., n,\ \ for\ i = 0, ..., n$$

Where **bool_to_int** is a function which takes as parameters two boolean arrays and the number of bits needed to encode the numbers (i.e. the length of the arrays). Returns:

- True if the integer encoding of the first array is less than the second
- False otherwise

3.3 Optimization

To search for the optimal solution, a custom search strategy was applied.
After a solution is found, we check if the maximum distance (objective function) is less than the minimum found already. If so, update the new best with the current solution.
A new constraint is added to the model at each iteration:

$$\bigvee_{i=0}^{n} \bigvee_{j=0}^{n} (\text{pred}[i][j] \neq \text{model}[\text{pred}[i][j]])$$

This allows to discard the last solution of the pred matrix and further explore the search space. This process is iterated until no further improvements are attained within 250 attempts (an empirically determined reasonable number), at which point we assume that the optimal solution has been reached.
The is an issue is with instance 7, where after 250 tries no better solution can be found. It stops before the time limit, thus labeling the solution as optimal, even though it is not (see table 5). Increasing the number of tries could not solve the problem.

3.4 Evaluation

To evaluate the model, two different solver algorithms have been tested:

- **CDCL** (Conflict-driven clause learning): uses a backtracking search with clause learning from conflicts, non-chronological backjumping, and unit propagation.
- **Local search (WSat)**: stochastic algorithm that starts with a random truth assignment and iteratively flips a variable in a randomly selected unsatisfied clause to minimize the number of unsatisfied clauses.

Moreover, the choice of encodings for the functions of the cardinality constraints at_most_one and exactly_one is important in the SAT solvers. Two different encodings have been tried:

- **Heule** encoding: uses a single auxiliary variable to represent the at_most_one constraint, and a binary variable to represent the exactly_one constraint. The encoding has a size of $O(n)$ and a propagation complexity of $O(n)$.
- **Bitwise** encoding: uses n binary variables to represent the at_most_one constraint, and n binary variables to represent the exactl_one constraint. The encoding has a size of $O(n^2)$ and a propagation complexity of $O(n^2)$.

Table 3 shows the result value of the objective functions for each solved instance, and each methodology applied.

Table 3: Objective function values for SAT model

	Heule encoding		Bitwise encoding	
ID	CDCL	WSat	CDCL	WSat
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	**244**	**244**	**244**	**244**
7	227	253	208	235
8	**186**	**186**	**186**	**186**
9	**436**	**436**	**436**	**436**
10	**244**	**244**	**244**	**244**
13	1800	1800	1358	1530

4 SMT Model

The SMT model follows a similar approach with respect to the other ones but it aims at exploiting theories such as *LIA* (Linear Integer Arithmetic theory) to reduce the search space of the problem. This feature is usually embedded in the SMT solver, in this case **Z3**, so the main task was to model the problem in

order to make the solver handle it properly.
Several models were tested including something similar to the *Model B* of the CP approach but at the end the simplest model was chosen since the others didn't lead to any improvement.

4.1 Decision Variables and Objective Function

For simplicity's sake, in the following lines, we will refer to m as the number of couriers and n as the number of items/customers.
There are just two sets of decision variables:

- **paths** $\rightarrow$ a three-dimensional list of boolean variables of size $m \times n \times n$ such that

$$path[i][j][k] = \begin{cases} True & \text{if j} \rightarrow \text{k is part of the path of courier i} \\ False & \text{otherwise} \end{cases}$$

- **num_visit** $\rightarrow$ a list of integers of size n representing the order in which the items are delivered within a path. It is necessary to implement the subtour elimination constraint with *Miller-Tucker-Zemlin formulation* [2].

The objective function **max_dist**, representing the maximum distance travelled among the couriers, is modelled as an integer decision variable. Its semantics is given by a set of constraints described in the next paragraph.

4.2 Constraints

1. The variables in *num_visit* should only take values between 0 and n-1 since a courier cannot visit more than n customers (one item per customer):

$$\forall i \in \{0, .., n-1\}\ (num\_visit[i]\ \geq\ 0)\ \wedge\ (num\_visit[i]\ \leq\ n-1)$$

2. We have to make sure that each customer is visited exactly once, this is achived by the following constraints (notice that boolean variables are treated as integers where the value *True* corresponds to 1 and *False* to 0):

$$\forall i \in \{0, .., m-1\}\, \forall k \in \{0, .., n-1\} \left(\sum_{j=0}^{n} paths[i][j][k] = 1 \right) \implies \left(\sum_{j=0}^{n} paths[i][k][j] = 1 \right)$$

$$\forall k \in \{0, .., n-1\} \left(\sum_{i=0}^{m-1} \sum_{j=0}^{n} paths[i][j][k] = 1 \right) \wedge \left(\sum_{i=0}^{m-1} \sum_{j=0}^{n} paths[i][k][j] = 1 \right)$$

3. Subtour elimination constraint in MTZ formulation:

$$\forall i \in \{0, .., m-1\}\, \forall j, k \in \{0, .., n-1\}\, paths[i][j][k] \implies (num\_visit[j] < num\_visit[k])$$

4. Each courier has a maximum load capacity:

$$\forall i \in \{0,..,m-1\} \left(\sum_{j=0}^{n} \sum_{k=0}^{n-1} paths[i][j][k] \times item\_sizes[k] \right) \leq load\_sizes[i]$$

where *item_sizes* is a list containing the *weights* of the items and *load_sizes* is a list containing the maximum load capacities of the couriers.

5. Each courier must exit the nodes they visit:

$$\sum_{i=0}^{m-1} \sum_{j=0}^{n} paths[i][j][j] = 0$$

6. Each path must begin and end at the depot:

$$\forall i \in \{0,..,m-1\} \left(\sum_{k=0}^{n-1} paths[i][n][k] = 1 \right) \wedge \left(\sum_{j=0}^{n-1} paths[i][j][n] = 1 \right)$$

7. Symmetry breaking constraint on lexicographical order of couriers with same load capacity, indeed if two couriers have same capacity then they are equivalent:

$$\forall j,k \in \{0,..,n-1\}\, \forall i_1, i_2 \in \{0,..,m-1\}\ (paths[i_1][n][j] \wedge paths[i_2][n][k]) \implies j < k$$

where $i_1 < i_2$ and $load\_sizes[i_1] = load\_sizes[i_2]$.

8. The set of constraints ensuring that the objective function has the intended meaning:

$$\forall i \in \{0,..,m-1\} \left(\sum_{j=0}^{n} \sum_{k=0}^{n} paths[i][j][k] \times distances[j][k] \right) \leq max\_dist$$

where *distances* is a matrix storing the distances between customers.

4.3 Evaluation

The model was tested on 21 instances of increasing complexity as for the other models and it was able to solve the first 10 of them. For bigger instances the solver returned *unknown* as result, meaning that it wasn't able to determine the satisfiability nor the unsatifiability of the given instance under the time limit of 300 seconds.
The results can be found in the table below (instances solved to optimality are in bold).

Table 4: Results for SMT model	
ID	Z3
1	**14**
2	**226**
3	**12**
4	**220**
5	**206**
6	**322**
7	168
8	**186**
9	**436**
10	**244**

5 MIP Model

The MIP model follows the same approach of the previous ones but it focuses on the mathematical formulation of the problem, creating a solution space that is restrained progressively by the constraints. For this reason, we exploited the mathematical part of [1] and we also took inspiration from the [3] to get better results. The input data is the same as in the other cases, but it's processed differently by the choice of library.

5.1 Environment

We chose Gurobi, a common library in python that provides optimal functionalities for solving MIP problems. We also considered to use ORtools and Pulp libraries but, after some tries, we noticed that Gurobi had optimal performance together with a simple implementation for the mathematical constraints.

5.2 Formulation

We started from the same formulation of the previous cases but we had to adapt it for the MIP model. In particular, we relied on the mathematical formulation and we added an additional decision variable to implement more constraints. We also decided not to use some of the previous decision variables, e.g. **pred** etc. because they were not necessary for this case.
It was crucial for simplicity purpose to consider, given the integer number of items and couriers from the instance file, the corresponding sets:

- $CUSTOMERS$ as $\{1, ..., items\}$.
- V is the set of all nodes, that is the same of $CUSTOMERS$ adding the depot in position 0, so we have $\{0, ..., items\}$.
- $COURIERS$ as $\{1, ..., couriers\}$.

5.3 Decision variables and objective function

We used a mathematical formulation exploiting three different decision variables, the first two essential for the majority of the constraints, the last one only for the MTZ one.

- x_{ijk} : it is a binary decision variable representing if a courier k covers the path from i to j. In practice, it's a 3D matrix of dimension *nodes* x *nodes* repeated $couriers$ times.
- y_{ik}: it is a binary decision variable representing if a courier k takes the item i.
- u_{ik}: it is a continuous decision variable representing the cost/distance value of the courier k. It is essential for the MTZ constraint to ensure that a courier k, going from i to j, has a value u_i lower than u_j; this ensures that the more a courier travels, the higher the distance.

For the objective function, we implemented the sum of all distances covered by all the couriers, enforcing to consider only the courier with the biggest one. This was implemented with a specific constraint that ensured to get this as a result, without reducing the solution space.

5.4 Constraints

For simplicity purpose, let's call:

1. $num\_couriers = m$
2. $num\_items = num\_customers = n$

Here are the constraints:

1. Each item must be assigned to only one courier, i.e. for each column j of the **courier** matrix, exactly one element in the row must be true.

$$\sum_{k=1}^{m} y_{ik} = 1 \quad \forall i \in CUSTOMERS$$

2. We have m couriers going out from the depot, that is in location 0; we used an implementation where every courier must be active.

$$\sum_{k=1}^{m} y_{0k} = m$$

3. Every courier must respect its maximum capacity.

$$\sum_{i=1}^{n} y_{ik} * item\_size[i] = load\_size[k] \quad \forall k \in COURIERS$$

4. A courier that goes from location $i \to j$, covers both i and j location, it is the only one covering that trait and also the opposite one can't be covered by someone else.

$$\sum_{j=0}^{n} x_{ijk} = \sum_{j=0}^{n} x_{jik} \quad \forall i \in CUSTOMERS \text{ and } \forall k \in COURIERS$$

5. If a courier goes from $i \to j$, it means that the corresponding decision variable must be true as well (equal to 1).

$$\sum_{j=0}^{n} x_{jik} = y_{ik} \quad \forall i \in CUSTOMERS \text{ and } \forall k \in COURIERS$$

6. The main diagonal of the x matrix must be equal to 0, indeed every courier can't go from a location j to the same location.

$$\sum_{j=0}^{n} x_{jjk} = 0 \quad \forall k \in COURIERS$$

7. Starting from an arbitrary node, every courier must go to another node.

$$\sum_{i=0}^{n} \sum_{k=1}^{m} x_{ijk} = 1 \quad \forall j \in CUSTOMERS$$

8. Every courier must go back to the depot.

$$\sum_{i=0}^{n} x_{i0k} = 1 \quad \forall k \in COURIERS$$

9. For every customer j, there must be only one vehicle k which goes to j from i.

$$\sum_{i=0}^{n} \sum_{k=1}^{m} x_{ijk} = 1 \quad \forall j \in CUSTOMERS$$

10. For every customer j, there must be only one vehicle k which leaves i to go to j.

$$\sum_{j=0}^{n} \sum_{k=1}^{m} x_{ijk} = 1 \quad \forall i \in CUSTOMERS$$

11. If a vehicle k visits location j, the same vehicle k should go from j to an arbitrary location i. The amount of couriers entering a node must be equal to the number of outgoing ones.

$$\sum_{i=0}^{n} x_{ijk} = \sum_{i=0}^{n} x_{ijk} \quad \forall k \in COURIERS \wedge \forall j \in V$$

12. MTZ formulation [2] for subtours' elimination.

$$u_{i,k} - u_{j,k} + n * x_{i,j,k} \leq n - 1$$
$$\forall i \in CUSTOMERS \wedge \forall j \in CUSTOMERS \quad \text{if} \quad i \neq j$$

5.5 Evaluation and results

On Gurobi, the optimization is performed by the optimize method. We tried different approaches to enhance the results exploiting the multi threading and similar solutions but they gave not remarkable results, so we kept the base line.

Table 5: Results for MIP model

ID	Gurobi	ID	Gurobi
1	**14**	10	**244**
2	**226**	12	447
3	**12**	13	424
4	**220**	16	286
5	**206**	19	334
6	**322**		
7	**167**		
8	**186**		
9	**436**		

6 Conclusion

After experimenting with the four approaches described in this work, we have noticed that:

- We could almost always solve to optimality every instance from 1 to 10, with exception of SAT for instance 7, SMT from 7, and for CP using Gecode solver.
- The SAT implementation, with the limit of only using boolean decision variables, required some workarounds like the *bool_to_int* function, which makes it very slow in generating the constraints for the biggest instances, taking several minutes just for that phase. Moreover the optimization strategy is very inefficient, and has some limitations.
- There are no errors detected by the *check_solution.py* script for any of the models.
- The instances from 11 to 21 were more difficult to solve due to the increased complexity, only CP and MIP gave some additional results for these cases but overall the complexity blocked the models.

References

[1] W. D. Fröhlingsdorf, "Using constraint programming to solve the vehicle routing problem with time windows," *University of Glasgow*, April 2018. Available at `http://www.jacktex.eu/research/material/master\_thesis.pdf`.

[2] AIMMS, "Miller-tucker-zemlin formulation," November 2020. Available at `https://how-to.aimms.com/Articles/332/332-Miller-Tucker-Zemlin-formulation.html`.

[3] B. S. F. Leite, "The vehicle routing problem: Exact and heuristic solutions," *Towards Data Science*, August 2023. Available at `https://towardsdatascience.com/the-vehicle-routing-problem-exact-and-heuristic-solutions-c411c0f4d734`.

Appendix A

Table A.1: succ decision matrix of Model A for Instance-3								
Location / Vehicle	1	2	3	4	5	6	7	8
1	1	2	8	4	6	3	7	5
2	1	8	3	2	5	6	7	4
3	8	2	3	4	5	6	1	7

Appendix B

Table B.1: Results for Model A using Gecode and Chuffed		
ID	Chuffed	Gecode
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	936
12	N/A	783
13	1912	1254
16	N/A	398
19	N/A	580
21	N/A	1134