CDMO Project - The Multiple Couriers Planning problem

Matteo Belletti matteo.belletti5@studio.unibo.it
Luca Morlino luca.morlino@studio.unibo.it
Ciprian Stricescu Razvan razvancipr.stricescu@studio.unibo.it

September 3, 2023

1 Introduction

The multiple courier routing problem (MCP) is a classic optimization problem in logistics and transportation. It is concerned with finding the shortest route to visit a set of customers with a fleet of vehicles, while satisfying certain constraints, such as the capacity of each vehicle and the distance between customers. In this report, we will investigate three different models for the MCP:

- Constraint programming (CP) model
- Satisfiability modulo theories (SMT) model
- Mixed-integer programming (MIP) model

More specifically we present two different models for both CP and SMT, as they represent two different approaches we decided to develop simultaneously in order to further compare the performance of the models. Although different, the input parameters and naming conventions are consistent across all models. This allows us to easily swap between models. Common input parameters are:

- **m** represents the number of couriers
- **n** represents the number of items
- **loadsizes** represents the vector of courier loads
- **sizes** represents the vector of item sizes
- **distances** represents the matrix of distances between items and between items and origin point

Moreover, within the CP and SMT approaches, we present two distinct models each, as they represent two parallel avenues we explored to further compare the performance of the models. Despite the differences in formulation and structure among these models, they all share a common objective: the minimization of the maximum distance traveled by a courier.
Despite the unique formulation of each model, we aimed to maintain a consistent base structure, ensuring compatibility among the approaches. The two general models we developed are:

- A **matrix-based** approach that leverages its inherent structure to address common challenges like sub-tour elimination efficiently.
- A **graph-based** approach, which offers flexibility in algorithm selection but entails greater complexity in both development and optimization.

Our inspiration for the graph-based approach was drawn from various papers discussing the application of graph theory to model and solve the MCP. We adapted these concepts to our context and benefited from examples provided by Minizinc[1]. Notably, the work of Masoumeh Vali and Khodakaram Salimifard[2] and Aitor Lopez Sanchez et al.[3] significantly influenced our graph-based approach.

Our exploration of a matrix-based approach stemmed from a desire to seek a simpler and more efficient method for problem modeling.

Through our analysis, we aim to offer a detailed overview of our models, methodology and results. In the following sections, we will delve into the graph methodology in detail, which is a common component across all our models. While we will not provide an exhaustive explanation of the matrix methodology, it will be included in our result comparisons for a comprehensive analysis.

The project took about six weeks to complete. Each of the different solution approaches was done by all members. In screen sharing when possible, in parallel otherwise. After an initial part of understanding the problem, a study of the papers concerning the MCP problem, and the choice of solving strategy we started with CP. Next we moved to SMT, and finally to MIP. Of course, there was no shortage of obstacles and continuous redefinition of constraints. The greatest difficulty was the translation of the constraints into the respective solution aproaches.

2 CP Model

In this section, we present two different methodologies for our constraint programming model, both implemented in Minizinc. These approaches use consistent naming conventions and data structures, and we define the common variables and parameters as follows:

ITEMS: An integer set of variables representing the items, defined as $1..n$.

COURIERS: An integer set of variables representing the couriers, defined as $1..m$.

DISTRIB_POINTS: An integer set of variables representing the distance matrix points, defined as $1..n+1$.

Variables and Parameters for the Graph Methodology:

GRAPH_NODES: An integer set of variables representing the nodes of the graph, defined as $1..n+2\cdot m$.

ORIGIN_NODES: An integer set of variables representing the origin and end points of the graph, defined as $n+1..n+2\cdot m$.

- ***START_ORIGIN_NODES***: An integer set of variables representing the origin nodes of the graph, defined as $n+1..n+m$.
- ***END_ORIGIN_NODES***: An integer set of variables representing the end nodes of the graph, defined as $n+m+1..n+2\cdot m$.

- ***distance_graph***: A matrix of shape $(n+2\cdot m)\times(n+2\cdot m)$ containing the distances between items, between items and origin points, and between origin and end points.
  - The distance between origin and end points is equal to 0.

2.1 Decision variables

In this section, we will explore the fundamental concept of decision variables, examining their definition, types, domains, and their respective roles within the context of the model presented.

Graph Methodology: We introduce a different set of decision variables to model the problem effectively:

- ***successor***: An array of variables of size GRAPH_NODES, representing the successor node for each node in the graph.
- ***predecessor***: An array of variables of size GRAPH_NODES, representing the predecessor node for each node in the graph.

- ***courier_per_item***: An array of variables of size GRAPH_NODES, associating couriers with items at each node in the graph.
- ***courier_count***: An array of variables of size ITEMS, indicating the count of couriers assigned to each item.
- ***load_at_node***: An array of variables of size GRAPH_NODES, with values ranging from 0 to max(load_sizes), representing the load at each node in the graph.

Node	Successor
1	3
2	6
3	7
4	2
5	1
6	5
7	4

Node	Predecessor
1	5
2	4
3	1
4	7
5	6
6	2
7	3

Figure 1: Example of Successor and Predecessor Arrays for seven items.

The "successor" array indicates the successor node for each node in the graph, while the "predecessor" array represents the predecessor node for each node in the graph. These arrays are essential for modeling the flow of couriers within the network.

2.2 Objective function

In this section, we define the objective function that we aim to minimize in our constraint programming model.

Within the Graph Methodology, the objective function is formulated as follows:

$$\textbf{Minimize: } \mathit{obj\_func} = z$$

Where:

- ***z*** represents the maximum distance traveled by any courier within the network.

The calculation of the objective function involves the following steps:

1. Calculate the distance traveled for each courier (*distance_done_per_courier*) by summing the distances between nodes in the *successor* array that belong to the same courier:

$$\mathit{distance\_done\_per\_courier}[\mathrm{c}] = \sum_{i \in \mathit{GRAPH\_NODES} \text{ where } \mathit{courier\_per\_item}[i]=c} \mathit{distance\_graph}[i, \mathit{successor}[i]]$$

2. Find the maximum distance traveled by any courier (z).

$$z = \max(\mathit{distance\_done\_per\_courier})$$

   The objective function is to minimize z, ensuring that no courier travels a distance greater than this value.

Additionally, we experimented with other objective functions, including:

1. **Alternative Objective Function 1:** This variant focuses on both the total distance and the maximum distance traveled by any courier. It aims to weight these factors accordingly in order to find an optimal solution.

$$\textbf{Minimize: } \mathit{obj\_func} = \mathit{total\_distance} \times 2 + (z - \min(\mathit{distance\_done\_per\_courier})) \times m$$

2. **Alternative Objective Function 2:** We also explored an objective function that considers the difference between the maximum and minimum distances traveled by couriers, along with the total distance. What we minimize is the sum of these values:

$$\textbf{Minimize: } \mathit{obj\_func} = (\max(\mathit{courier\_distances}) - \min(\mathit{courier\_distances})) + \ \mathit{total\_distance}$$

These alternative objective functions were experimented with to evaluate their impact on the optimization results.

2.3 Constraints

We define the constraints used in the Graph Methodology as follows:

1. **Initialization Constraints:**

   (a) Ensure that the predecessor of start nodes corresponds to end nodes:

   $$\begin{aligned} \forall i \in (n+2..n+m)\colon \text{predecessor}[i] &= i+m-1 \\ \text{predecessor}[n+1] &= n+2\cdot m \end{aligned}$$

   (b) Associate each end node with its corresponding start node:

   $$\begin{aligned} \forall i \in (n+m+1..n+2\cdot m-1)\colon \text{successor}[i] &= i-m+1 \\ \text{successor}[n+2\cdot m] &= n+1 \end{aligned}$$

   (c) Associate each start and end node with a vehicle:

   $$\begin{aligned} \forall i \in \text{START\_ORIGIN\_NODES}\colon \text{courier\_per\_item}[i] &= i-n \\ \forall i \in \text{END\_ORIGIN\_NODES}\colon \text{courier\_per\_item}[i] &= i-n-m \\ \forall i \in \text{ITEMS}\colon \text{courier\_count}[i] &= \text{courier\_per\_item}[i] \end{aligned}$$

   (d) Set the initial load at depot nodes to zero:

   $$\forall i \in \text{START\_ORIGIN\_NODES}\colon \text{load\_at\_node}[i] = 0$$

2. **Predecessor/Successor Constraints:**

   (a) Ensure that predecessor and successor nodes are correctly linked:

   $$\forall i \in \text{GRAPH\_NODES}\colon \text{successor}[i] = \text{predecessor}[\text{predecessor}[i]]$$

   (b) Eliminate sub tours:

   $$\begin{aligned} & \text{circuit(successor)} \\ & \text{circuit(predecessor)} \\ \text{Constraint:}\quad & \text{circuit}(x) \\ \text{Where:}\quad & x[i] = j \quad \text{means that } j \text{ is the successor of } i. \end{aligned}$$

3. **Vehicle Assignment and Load Constraints:**

(a) Assign the same vehicle to each item as its predecessor:

$$\forall i \in \text{ITEMS}\colon \text{courier\_per\_item}[\text{predecessor}[i]] = \text{courier\_per\_item}[i]$$
$$\forall i \in \text{ITEMS}\colon \text{courier\_per\_item}[\text{successor}[i]] = \text{courier\_per\_item}[i]$$

(b) Ensure that each courier is assigned to at least one item:

$$\forall c \in \text{COURIERS}\colon \text{count\_eq}(\text{courier\_count}, c) \geq 1$$

(c) Maintain load consistency as items are delivered:

$$\forall i \in \text{ITEMS}\colon \text{load\_at\_node}[i] + \text{weight}[i] = \text{load\_at\_node}[\text{successor}[i]]$$
$$\forall i \in \text{START\_ORIGIN\_NODES}\colon \text{load\_at\_node}[i] = \text{load\_at\_node}[\text{successor}[i]]$$

(d) Ensure that partial load does not exceed the vehicle capacity for each item:

$$\forall i \in \text{ITEMS}\colon \text{load\_at\_node}[i] \leq \text{load\_sizes}[\text{courier\_per\_item}[i]]$$

(e) Ensure that the final load does not exceed the vehicle capacity for each courier:

$$\forall c \in \text{COURIERS}\colon \text{load\_at\_node}[c + n + m] \leq \text{load\_sizes}[c]$$

4. **Objective Function:**

(a) Compute the distance traveled by each courier and store it in `distance_done_per_courier`:

$$\forall c \in \text{COURIERS } (\text{distance\_done\_per\_courier}[c] =$$

$$\textstyle\sum_{i \in \text{GRAPH\_NODES where courier\_per\_item}[i]=c} \text{distance\_graph}[i, \text{successor}[i]])$$

We added redundant constraints to the model in order to improve the performance of the solver. Redundant constraints are constraints that can be removed from the model without changing the set of feasible solutions. However, they can sometimes improve the performance of the solver by helping the solver to find a feasible solution more quickly. In our case, adding redundant constraints resulted in an improvement in the performance of the solver.

Symmetry Breaking:
To break symmetry in the model, we employ lexicographic (lex) constraints. These constraints ensure a specific order or assignment, eliminating symmetrically equivalent solutions from the search space.

- **Node Ordering Symmetry Breaking:**
  Let *node_order* be an array of variables representing the order in which nodes are visited, with *node_order*$[i]$ denoting the position of node i. The lexicographic constraint is defined as follows:

  $$\text{constraint lex\_less}(\mathit{node\_order}, [\mathit{successor}[i] \mid i \in 1..n])$$

  This constraint enforces that the values in *node_order* must be lexicographically less than the values of their corresponding *successor* variables, ensuring a specific order of node visits.

- **Vehicle Assignment Symmetry Breaking:**
  We introduce an array of variables *vehicle_assignment* to represent the assignment of nodes to vehicles, with *vehicle_assignment*$[i]$ indicating the vehicle assigned to node i. The lexicographic constraint for vehicle assignment is defined as:

  $$\text{constraint lex\_less}([\mathit{vehicle\_assignment}[i] \mid i \in 1..n], [\mathit{vehicle\_assignment}[i] \mid i \in 2..n])$$

  This constraint enforces lexicographic ordering of vehicle assignments, breaking symmetry in vehicle assignments.

By employing these symmetry-breaking constraints, we guide the solver to prioritize solutions that adhere to the specified order or assignment, effectively eliminating symmetrically equivalent solutions from consideration.

2.4 Validation

To validate our models, we implemented them in MiniZinc and executed them using two different solvers, Gecode and Chuffed. We employed various search and restart strategies to assess their performance. The validation setups for both the Matrix and Graph models are as follows:

- **For the Matrix Model:**
  - *int_search* in combination with *restart_linear*, with scale $n \cdot n$.
- **For the Graph Model:**
  - *int_search* in combination with *restart_luby*, with scale $n \cdot n$, without symmetry breaking constraints.
  - *int_search* in combination with *restart_luby*, with scale $n \cdot n$, with symmetry breaking constraints.

All experiments were conducted on a computer with the following specifications: AMD Ryzen 5 2600X Six-Core Processor 3.60 GHz and 16GB of RAM.

The software environment for our experiments included: Windows 10 and Minizinc Version 2.7.6.

All the solvers had a time limit of 300 seconds.

Here we present the results for each combination on instances from 8 to 12:

ID	Gecode w/out SB	Chuffed w/out SB	Gecode + SB	cHUFFED + SB
8	186	**186**	186	186
9	436	436	436	436
10	244	244	244	N/A
11	148	1441	N/A	N/A
12	868	1211	N/A	N/A

Table 1: Results on the graph model using both Gecode and Chuffed solvers with and qithout symmetry breaking.

ID	Gecode	Chuffed
8	186	**186**
9	N/A	436
10	N/A	244
11	N/A	N/A
12	N/A	N/A

Table 2: Results on the matrix model using both Gecode and Chuffed solvers.

As we can see, the results vary between the two models. Generally, the matrix model performs faster on smaller instances, however, it encounters significant challenges when attempting to tackle larger, more intricate instances. In contrast, the graph model, with its greater complexity, exhibits a slower runtime on smaller instances it runs a wider range of problem sizes, albeit without achieving optimal solutions.

It's worth noting that our experimentation with different solvers also reveals noteworthy insights. Chuffed, in particular, appears to excel in efficiently solving instances across various sizes, enhancing the overall performance of both models.

3 SMT Model

3.1 Decision variables

- *graph_nodes*: this set of variables is structured as a three-dimensional boolean matrix denoted as $i \times j \times k$, where i and j correspond to the number of items plus 1 (the origin) and k represents the number of couriers involved in the problem. In practice, each courier has an adjacency matrix where the point (i,j) has the value *true* if the courier travels the edge from i to j. The last raw and the last column belong to the origin.

Each element within this matrix is uniquely labeled as $x_{i,j,k}$. These variables serve to establish an adjacency matrix that delineates the connectivity and traversal possibilities for couriers concerning items and locations.

- *max_distance_per_courier* is a list of k integers, where k represents the number of couriers. Each element of this list corresponds to the total distance traveled by an individual courier.
- *mtz_matrix* is a list of n integers. Each integer variable is named *mtz_i*," where i is an integer ranging from 0 to n (inclusive). n represents the number of delivery points.

3.2 Objective function

In SMT we try to minimize the maximum value of the *max_distance_per_courier* list. At every step of the loop, the model checks if the new maximum value is lower then the best one. If this is true, we add a new constraint to the model requiring the *max_distance* to be less than the previous one.

3.3 Constraints

- To enforce that there are no self-loops within this graph, meaning that no node is connected to itself, we introduce the following constraint. To get this, in the *graph_nodes* matrix, every courier k has a adjacency matrix where the diagonal is equal to 0.

$$\forall k \text{ in } [0, m-1] : \sum_{i=0}^{n} \text{graph}[i][i][k] = 0$$

- The next constraint imposes that each node is visited only once. To do this, two constraints Pbec(Pseudo-Boolean Equality Constraint) are placed in logical and. In practice, given i, the sum of all rows i of all couriers must be equal to 1. This would mean that only one courier passes through distribution point i. Obviously this must not concern the last column and the last row as all couriers must start from the origin and arrive at the origin.

$$\forall i \text{ in } [0, n-1] :$$
$$\sum_{j=0}^{n} \sum_{k=0}^{m-1} \text{graph\_nodes}[i][j][k] \times 1 = 1$$

AND

$$\sum_{j=0}^{n} \sum_{k=0}^{m-1} \text{graph\_nodes}[j][i][k] \times 1 = 1$$

- The origin node can have only one arrival edge and one departure edge. The last row n and last column n of each courier adjacency matrix must have only one value *true*

$$\forall k \text{ in } [0, m-1] :$$
$$\sum_{j=0}^{n-1} \text{graph\_nodes}[n][j][k] \times 1 = 1$$

AND

$$\forall k \text{ in } [0, m-1]:$$
$$\sum_{j=0}^{n-1} \text{graph\_nodes}[j][n][k] \times 1 = 1$$

- The load of each courier can not exceed its load size. To get this, the sum of all of the elements of the adjacency matrix set to *true* multiplied by their respective size must not exceed the load size of that courier

$$\forall k \text{ in } [0, m-1]:$$
$$\sum_{i=0}^{n-1} \sum_{j=0}^{n} \text{graph}[i][j][k] \times \textit{item\_size}[i] \leq \text{load\_size}[k]$$

- The next constraint is very important as it imposes that each node has the same number of incoming and outgoing edges. The number of *true* values in a row of a courier must be equal to the number of the *true* values in a column.

$$\forall k \text{ in } [0, m-1], \forall j \text{ in } [0, n]:$$
$$\sum_{i=0}^{n} \text{graph\_nodes}[i][j][k] = \sum_{i=0}^{n} \text{graph\_nodes}[j][i][k]$$

- We have to avoid sub-tours in a single path courier. To get this we add a list of decision variables and constraints to ensure that the resulting tour does not contain sub-tours. Miller-Tucker-Zemlin subtour elimination algorithm constructs a new list of decision variables that assign each distribution point a value. In a courier route, each node must have a value that is always greater than that of the previous node. Doing so avoids reaching a node that has already been traversed.

$$\forall k \text{ in } [0, m-1], \forall i \text{ in } [0, n-1], \forall j \text{ in } [0, n-1]:$$
$$\text{mtz\_matrix}[i] + \text{graph\_nodes}[i][j][k] \leq \text{mtz\_matrix}[j] + (n-1) \times (1 - \text{graph\_nodes}[i][j][k])$$

3.4 Validation

3.4.1 Experimental design

We used a custom search strategy. The model is runned a x number of times. We empirically set x to 1000. Each time a solution that satisfies the constraints is found, we add a constraint that sets the new maximum path. For the hardware specifications read the sub-section 2.4.

3.4.2 Experimental results

The results of the SMT approach are shown on the table 2.

4 MIP Model

Our Mixed Integer Programming model is implemented using 'ortools' Python's library. Google Optimization Tools (a.k.a., OR-Tools) is an open-source, high-performance optimization library developed by Google.

When considering the MIP Model, we implemented the Graph-based approach using adjacency matrices to represent the aforementioned graphs.

ID	Z3
1	**14**
2	**226**
3	**12**
4	**220**
5	**206**
6	**322**
8	**186**
9	436
10	**244**

Table 3: SMT results

4.1 Decision variables

- *max_distance*: A positive integer, with lower bound equal to zero and upper bound to infinity. It's the variable which we minimize in the objective function.
- *assignment*: A 3D matrix of size *num_couriers* x (*num_items* + 1) x (*num_items* + 1), where $assignment[i][j][k] = 1$ if and only if courier i goes from distribution point j to k, implying he loads both item j and k.
- *mtz_matrix*: a 2D matrix of size *num_couriers* x (*num_items* + 1), where each line i ensures correct routing of the corresponding courier as explained in the constraints below.

4.2 Objective function

As the objective function of the MIP model we ask for the minimization of the variable 'max_distance'. The variable is declared as an Integer variable between zero and infinity and later bound by all of the couriers's travelled distances, thus fixing a lower bound.

Once this is done the meaning of the objective function is straightforward: minimizing the variable with set lower bound, and upper bound = infinity.

4.3 Constraints

1. Ensures that the assignment matrix *assignment* has diagonal elements ($assignment[i][j][j]$) always equals to zero, as they represent the path from a distribution point to itself.

$$\sum_{i=1}^{num\_couriers} \sum_{j=0}^{num\_items} assignment[i][j][j] = 0$$

2. Each item is assigned to exactly one courier.

$$\forall n \in \text{num\_items} : \sum_{i=0}^{\text{num\_couriers}-1} \sum_{k=0}^{\text{num\_items}} \text{assignment}[i][n][k] = 1$$

$$\forall n \in \text{num\_items} : \sum_{i=0}^{\text{num\_couriers}-1} \sum_{k=0}^{\text{num\_items}} \text{assignment}[i][k][n] = 1$$

3. Each courier's path length must be lower or equal compared to $max_distance$, thus setting the lower bounds.

$$\forall c \in \text{num\_couriers}:$$
$$\sum_{j=0}^{num\_items} \sum_{k=0}^{num\_items} \text{assignment}[c][j][k] \cdot \text{item\_distances}[j][k] \leq \text{max\_distance}$$

4. Each courier's path must have a starting point and thus its assignment matrix must have at least one of the element of the last row equal to one as it must bring at least one item.

$$\forall c \in \text{num\_couriers}:$$
$$\sum_{k=0}^{num\_items} assignment[c][num\_items][k] = 1$$

5. For each courier i, the total size of its assigned items must be lower than its capacity $courier\_capacities[i]$.

$$\forall c \in \text{num\_couriers}:$$
$$\sum_{j=0}^{num\_items-1} \sum_{k=0}^{num\_items} assignment[i][j][k] \cdot item\_sizes[j] \leq courier\_capacities[i]$$

6. The number of incoming and outgoing arcs must be equal so that along the path all distribution points are connected correctly.

$$\forall c \in \text{num\_couriers}:$$
$$\forall n \in \text{num\_items}:$$
$$\sum_{k=0}^{num\_items} assignment[c][n][k] = \sum_{k=0}^{num\_items} assignment[c][n][k]$$

7. I use the MTZ algorithm to deal with subtours, linking together the *mtz_matrix* and the *assignment* matrix forcing a fully connected start-to-origin path.

$$\forall c \in \text{num\_couriers}:$$
$$\forall n \in \text{num\_items}:$$
$$\forall k \in \text{num\_items}:$$
$$if(n \neq num\_items):$$
$$u[i][j] - u[i][k] + 1 \leq (num\_items - 1) \cdot (1 - assignment[i][j][k])$$

4.4 Validation

4.4.1 Experimental design

To validate the MIP model we wrote a python script opening the specified instance (single instances or range) and, through a function, using the ortools script to get a solution within the five minutes limit. The script automatically saved the solution in a .json file that we validated with the provided 'checksolution.py'. For the hardware specifications read the sub-section 2.4.

4.4.2 Experimental results

The results of the MIP approach are shown on the table 3.

ID	OR-Tools
1	**14**
2	**226**
3	**12**
4	**220**
5	**206**
6	**322**
8	**186**

Table 4: MIP results

5 Conclusions

In the project we tackled a version of the well known multiple-couriers problem. We used different technologies exploiting constraint programming with minizinc, SMT and MIP. Depending on the technology used, we obtained different results, demonstrating how different solution lead to different results. In our case the minizinc version was the most succesful solving of the instances. In SMT and MIP we solved different percentages of the given instances. It could be both related to the approach used and to the solution adopted. For future developments, we will try to further optimize our solutions and perform more comparisons on a pool of instances with different complexity. This would lead to a better understanding of the languages capabilities.

References

[1] link

[2] Akün, Ö. (2017). *Multi-objective Traveling Salesman Problem and Its Variants: A Survey*. ModRef 2017. link

[3] Sánchez, A. L., Lujak, M., Semet, F., & Billhardt, H. *On balancing fairness and efficiency in routing of cooperative vehicle fleets*. link