Combinatorial Decision Making and Optimization

Filippo Teodorani, Diego Rossi
University of Bologna
Email: filippo.teodorani@studio.unibo.it, diego.rossi6@studio.unibo.it

June 8, 2025

1 Introduction

This report contains the strategies, the results and the models we implemented for solving the MCP problem. The approaches involved are: Constraint Programming (CP), Satisfiability Modulo Theories (SMT), and Mixed-Integer linear Programming (MIP).
Our goal was to develop effective models for solving the MCP problem. The results of our experiments will be used to evaluate the performance of the different approaches and discuss the limitations of our work.

1.1 Group Work

Our work was split in two parts: Diego Rossi focused mainly on CP while Filippo Teodorani implemented SMT and MIP. We independently performed testing and validation for the models that we created on the different programming languages on our personal machines, equipped with Intel i7 processors and 16GB of RAM both running Windows. It took us a total of 3 to 4 months to complete the whole project implementation.

1.2 Common Parameters

Parameters that are present in the .dat format instances are common to all different models, programming languages and optimization strategies and define the problem's context and properties.

- Number of couriers *(m)*: number of available couriers for the item distribution.
- Number of items *(n)*: number of items to be distributed (n¿m).
- Maximum load capacities *(capacity)*: maximum load size for each courier.
- Item sizes *(item_size)*: sizes of each item.

- Distance matrix *(distance_matrix)*: matrix that presents the distances between each customer locations and the origin.

1.3 Objective Value

The ultimate goal of the MCP problem is to minimize the maximum distance traveled by any courier during the delivery of the customer's items.

$$\text{obj} = \min \max_{k \in \{1,\dots,m\}} \sum_i \sum_j \text{dist}_{i,j} x_{i,j,k}$$

2 CP Model

We have used MiniZinc, a free and open source high-level constraint modeling language, to address the MCP problem. As it is not possible to save the output results from the MiniZinc IDE, a simulation of the solving process was implemented in Python, by means of the *subprocess* module.
We have implemented the Multiple Couriers Planning problem with three variants: "basemodel" (representing the first model we built), "popenmodel" (an alternative variant using a different search strategy and the class Popen of the subprocess module to save intermediate unoptimal results) and "lastmodel_sb" (the most complex CP model we've built that uses symmetry breaking constraint and the class Popen too).
The three models use the same decision variables and objective function, differing only in the solver search configuration and the symmetry breaking constraint implementation. All variables and domains below apply to all variants.

2.1 Specific Parameters

In addition to the parameters listed in section 1.2, these parameters are implemented in the CP models to simplify the optimization process: in particular, we implemented upper and lower bounds for two different variables, in order to restrict the search space.

- Minimum path length *(min_path)*: specifies the lower limit for feasible path lengths.
- Maximum path length *(max_path)*: specifies the upper limit for feasible path lengths.
- Minimum number of packs *(min_packs)*: indicates the least number of packs that a courier is required to deliver.
- Maximum number of packs *(max_packs)*: indicates the greatest number of packs that a courier is allowed to deliver.

The core concept behind defining these upper and lower bounds is to ensure that the triangle inequality holds for the graph represented by the distance matrix and to establish valid bounds on the distance, leveraging the information about the number of packs.

2.2 Decision Variables

Below are the decision variables in our MiniZinc models, along with their domains and intended meanings:

- *route*: a two-dimensional matrix of size $m \times (\text{max packs} + 2)$, containing integer values from 1 to $n + 1$, used to specify the destination of courier j at each point in time.
- *courier_distance*: an integer array of size m, with values ranging from 1 to max path. The array `courier_distance_j` stores the distance that each courier j is required to travel.
- *packs*: an array of size n, with values ranging from 1 to m, indicating which courier is assigned to deliver each package. For example, `packs_i` $= j$ means that courier j is responsible for delivering package i.
- *max_distance*: assignment of value of the objective function to minimize.

Although not directly optimized, we calculate the total load of each courier to aid in symmetry breaking.

$$load_j = \sum_{i=1}^{n} (packs_i = j) \cdot item\_size_i.$$

When two couriers are structurally similar (same capacity, etc.), we enforce an ordering based on increasing load and distance to reduce symmetrical solutions in the search space.

2.3 Objective Function

The objective function to minimize is the max_distance function, referenced in section 1.3, as it's common for multiple approaches.

2.4 Constraints

The Constraint Programming model is implemented using a variety of constraints. To make visualization clearer, we write

$$L = \texttt{max\_path\_length}$$

The following list contains all the constraints that have been defined.

1. each courier starts and ends at the origin.

$$\forall j \in \{1, ..., m\} route_{j,1} = origin \ \wedge route_{i,max\_path\_length} = origin$$

2. origin isn't reached except in the intermediate positions.

$$\forall j \in \{1, ..., m\}, \forall i \in \{2, ..., min\_packs + 1\} : route_{j,i} \neq origin.$$

3. the load capacity of the couriers is respected:

$$\forall j \in \{1, \ldots, m\} : \quad \sum_{i=2}^{L-1} \begin{cases} \texttt{item\_size}[\text{route}_{j,i}], & \text{route}_{j,i} \leq n, \\ 0, & \text{otherwise} \end{cases} \leq \text{capacity}_j.$$

4. in each position, two couriers can't deliver the same item at the same time:

$$\forall i \in \{2, \ldots, L-1\} : \quad \big\{\text{route}_{1,i}, \ldots, \text{route}_{m,i}\big\} \setminus \{\text{origin}\} \text{ are all different.}$$

5. coherence between route and packs ($packs[i] = j \iff i \in$ path of j):

$$\big(i \in \{\text{route}_{j,2}, \ldots, \text{route}_{j,L-1}\}\big) \iff (packs[i] = j)$$

$$\forall i \in \{1, \ldots, n\}, \ \forall j \in \{1, \ldots, m\}.$$

6. if a courier goes back to origin in position i, then for all the successive positions remains at the origin:

$$\forall j, \ \forall i \in \{2, \ldots, L-1\} : \quad \text{route}_{j,i} = \text{origin} \implies$$

$$\forall k \in \{i, \ldots, L-1\} : \text{route}_{j,k} = \text{origin}.$$

7. global cardinality closed constraint: ensures that, across all routes, each node (including the depot and each delivery point) appears exactly the number of times specified by the corresponding entries.

8. distance ran by each courier gets computed:

$$\forall j \in \{1, \ldots, m\} : \quad \text{courier\_distance}_j = \sum_{i=2}^{L} d\big(\text{route}_{j,i-1}, \text{route}_{j,i}\big),$$

9. Symmetry Breaking constraint:

$$\text{We define } load_j = \sum_{i:packs[i]=j} item\_size[i].$$

$$\forall (j_1, j_2) \in \texttt{similars}, \ j_1 < j_2 : \quad load_{j_1} \leq load_{j_2} \ \wedge$$

$$\text{courier\_distance}_{j_1} \leq \text{courier\_distance}_{j_2}.$$

2.5 Validation

We collected results for three different models that we have implemented, with the first one that was tested both on Gecode and Chuffed solvers, while the other two models were only tested using Gecode.
Two specific strategies we have implemented to upgrade our models' performances were ordering the capacities array in decreasing order (to fill the couriers that have a higher capacity before) and then back to their original disposition and the use of the class Popen of the module subprocess to save intermediate results in the .JSON file.
Moreover, for both 'popenmodel' and 'lastmodel_sb' specific search strategies were adopted, in particular we have used Large Neighbourhood Search techniques such as int_search, relax_and_reconstruct and restart_geometric commands in MiniZinc.

The following table shows all the results for the implemented models:

Instance	basemodel (Chuffed)	basemodel (Gecode)	popenmodel	lastmodel_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	167	167	167	167
8	186	186	186	186
9	436	436	436	436
10	244	244	244	244
11	N/A	N/A	N/A	N/A
12	N/A	N/A	N/A	N/A
13	N/A	N/A	486	504
14	N/A	N/A	N/A	N/A
15	N/A	N/A	N/A	N/A
16	N/A	N/A	286	N/A
17	N/A	N/A	N/A	N/A
18	N/A	N/A	N/A	N/A
19	N/A	N/A	334	N/A
20	N/A	N/A	N/A	N/A
21	N/A	N/A	N/A	N/A

3 SMT Model

This section presents the SMT-based model used to solve the multi-courier delivery routing problem. The model is implemented using the Z3 SMT solver under the theory of linear integer arithmetic.

3.1 Decision variables

The model defines the following decision variables:

- **Routing Variables**:

$$\texttt{table}_{k,i,j} \in \{\text{true}, \text{false}\}$$

  Boolean variables indicating whether courier k travels directly from location i to location j.

- **Courier Distance Variables**:

$$\texttt{courier\_distance}_k \in \mathbb{Z}_{\geq 0}$$

  Total distance traveled by courier k, constrained within $[0, \texttt{max\_path}]$.

- **Sub-tour Elimination Variables**:

$$u_{k,i} \in \mathbb{Z}_{\geq 0}$$

  Auxiliary integer variables used for Miller-Tucker-Zemlin (MTZ) sub-tour elimination, bounded by $[0, n-1]$, where n is the number of delivery locations (excluding the depot).

- **Objective Variable**:

$$\texttt{obj} \in \mathbb{Z}_{\geq 0}$$

  Represents the longest distance traveled by any courier, which is minimized.

3.2 Objective function

The goal is to minimize the maximum distance traveled among all couriers:

$$\min \texttt{obj}$$

Subject to:

$$\texttt{obj} \geq \texttt{courier\_distance}_k \quad \forall k \in \{0, \ldots, m-1\}$$

Each courier's distance is computed as:

$$\texttt{courier\_distance}_k = \sum_{i=0}^{n} \sum_{j=0}^{n} \texttt{table}_{k,i,j} \cdot \texttt{dist}_{i,j}$$

3.3 Constraints

Flow Conservation Each visited node must be entered and exited exactly once by the same courier:

$$\sum_j \texttt{table}_{k,i,j} = \sum_j \texttt{table}_{k,j,i} \quad \forall k, i$$

Visit Coverage Each delivery location (excluding the depot) must be visited exactly once:

$$\sum_{k,i} \texttt{table}_{k,i,j} = 1 \quad \forall j < n$$

Depot Constraints Each courier must start and end at the depot (node n):

$$\sum_j \texttt{table}_{k,n,j} = 1, \quad \sum_j \texttt{table}_{k,j,n} = 1$$

Capacity Constraints Couriers must respect their individual load capacities:

$$\sum_{i,j<n} \texttt{table}_{k,i,j} \cdot \texttt{size}_j \leq \texttt{max\_load}_k$$

Visit Count Bounds Each courier is constrained to serve between `min_packs` and `max_path` delivery points:

$$\texttt{min\_packs} \leq \sum_{i,j<n} \texttt{table}_{k,i,j} \leq \texttt{max\_path}$$

Symmetry Breaking To reduce redundant symmetric solutions, the model forbids courier paths that include both directions between two locations:

$$\neg(\texttt{table}_{k,i,j} \wedge \texttt{table}_{k,j,i}) \quad \forall i \neq j, i, j < n$$

Sub-tour Elimination To eliminate cycles not connected to the depot (subtours), MTZ-style constraints are used:

$$u_{k,j} \geq u_{k,i} + 1 - n \cdot (1 - \texttt{table}_{k,i,j}) \quad \forall k, \forall i \neq j < n$$

3.4 Validation

The model was implemented using Python and Z3. The following setup was used for validation:

- **Solver**: Z3 (Python API)
- **Search Strategy**: Z3's default SMT solving
- **Hardware**: Intel i7, 16GB RAM
- **Timeout**: 300 seconds per instance

The obtained results for the SMT model are:

Instance	SMT
1	14
2	226
3	12
4	220
5	206
6	322
7	167
8	186
9	436
10	244

Table 1: Results using z3 with simmetry breaking constraints

4 MIP Model

This section describes the Mixed Integer Programming (MIP) model developed to solve the multi-courier delivery routing problem. The model is implemented using the `mip` Python package and includes standard routing constraints, capacity limitations, and sub-tour elimination using a variant of the Miller-Tucker-Zemlin (MTZ) formulation.

4.1 Decision Variables

The model uses the following integer variables:

- **Routing Variables**:

$$\texttt{table}_{k,i,j} \in \{0, 1\}$$

  Binary variable equal to 1 if courier k travels from location i to location j, 0 otherwise.

- **Courier Distance Variables**:

$$\texttt{courier\_distance}_k \in \mathbb{Z}_{\geq 0}$$

  Total travel distance for courier k, computed as a sum of edge distances.

- **Sub-tour Elimination Variables**:

$$u_{k,i} \in [1, n]$$

  Integer variables used to prevent sub-tours for courier k, where n is the number of locations (including the depot).

- **Objective Variable**:

$$\texttt{obj} \in \mathbb{Z}_{\geq 0}$$

  The maximum distance traveled by any courier, minimized in the objective.

4.2 Objective Function

The goal is to minimize the longest route among all couriers:

$$\min \texttt{obj}$$

Subject to:

$$\texttt{obj} \geq \texttt{courier\_distance}_k \quad \forall k \in \{0, \ldots, m-1\}$$

Courier distances are computed as:

$$\texttt{courier\_distance}_k = \sum_{i=0}^{n} \sum_{j=0}^{n} \texttt{table}_{k,i,j} \cdot \texttt{dist}_{i,j}$$

4.3 Constraints

Flow Conservation Each courier must maintain a balanced flow at each node:

$$\sum_{j=0}^{n} \texttt{table}_{k,i,j} = \sum_{j=0}^{n} \texttt{table}_{k,j,i} \quad \forall k \in \{0,\ldots,m-1\},\ i \in \{0,\ldots,n\}$$

Unique Visit Each delivery location (excluding the depot) is visited exactly once by exactly one courier:

$$\sum_{k=0}^{m-1} \sum_{i=0}^{n} \texttt{table}_{k,i,j} = 1 \quad \forall j \in \{0,\ldots,n-1\}$$

Depot Constraints Each courier must depart from and return to the depot node n:

$$\sum_{j=0}^{n-1} \texttt{table}_{k,n,j} = 1, \quad \sum_{i=0}^{n-1} \texttt{table}_{k,i,n} = 1 \quad \forall k$$

Capacity Constraints The total load carried by each courier cannot exceed its capacity:

$$\sum_{i=0}^{n} \sum_{j=0}^{n-1} \texttt{table}_{k,i,j} \cdot \texttt{size}_j \leq \texttt{max\_load}_k \quad \forall k$$

Visit Count Bounds Each courier must serve at least `min_packs` and at most `max_packs` locations:

$$\texttt{min\_packs} \leq \sum_{i=0}^{n} \sum_{j=0}^{n-1} \texttt{table}_{k,i,j} \leq \texttt{max\_packs} \quad \forall k$$

Symmetry Breaking (Optional) To reduce equivalent symmetric routes, enforce at most one direction between any pair of nodes:

$$\texttt{table}_{k,i,j} + \texttt{table}_{k,j,i} \leq 1 \quad \forall k,\ \forall i \neq j,\ i,j \in \{0,\ldots,n-1\}$$

Sub-tour Elimination (MTZ Formulation) Prevent sub-tours using auxiliary variables $u_{k,i}$:

$$u_{k,j} \geq u_{k,i} + 1 - n \cdot (1 - \texttt{table}_{k,i,j}) \quad \forall k,\ \forall i \neq j,\ i,j \in \{0,\ldots,n-1\}$$

4.4 Validation

The MIP model is implemented using the `mip` Python package, with CBC as the default solver. The following setup was used:

- **Solver**: CBC via Python-MIP+
- **Hardware**: Intel i7 CPU, 16GB RAM
- **Timeout**: 300 seconds per instance

The obtained results for the MIP model are:

Instance	CBC
1	14
2	226
3	12
4	220
5	206
6	322
7	185
8	186
9	436
10	244

Table 2: Results using z3 with simmetry breaking constraints

5 Conclusions

When evaluating various techniques for solving optimization problems, it becomes evident that each method brings its own advantages and limitations, and their suitability depends on the nature of the specific task at hand.
Constraint Programming is very flexible and can quickly find solutions to model complex problems with relative ease. It is particularly effective when global constraints are available, as is the case for the MCP problem. However, for less researched challenges, this can be more complex.
On the other hand, Mixed-Integer Programming (MIP) is valued for its straightforward formulation and computational speed. Nevertheless, the necessity to represent problems through a large number of variables and constraints can lead to high memory usage, and its reliance on linear models can be restrictive for non-linear or highly combinatorial tasks.
Satisfiability Modulo Theories (SMT) solvers, while powerful in reasoning about logical constraints, are generally not optimized for numerical optimization. Encoding optimization problems into a form suitable for SMT solvers often introduces additional complexity and inefficiency. Memory limitations and less effective optimization support make them less ideal for problems like the Vehicle Routing Problem.
In conclusion, the most appropriate solving technique depends on the problem's structure, complexity, and the availability of domain knowledge. Making an informed choice involves balancing factors such as efficiency, scalability, model clarity, and solver capabilities.