Report

Baytcheva Christina - christina.baytcheva@studio.unibo.it
Di Iulio Anna - anna.diiulio@studio.unibo.it
Piscopo Alessandro - alessandro.piscopo@studio.unibo.it

November 20, 2024

1 Introduction

In this report we aim to illustrate how we solved the common task defined as the multiple courier problem. We will illustrate how we tackled this issue, each in their assigned models. We split the work in the following way:

- CP: Christina Baytcheva
- SMT: Anna Di Iulio
- MIP: Alessandro Piscopo

1.1 Common parts

- **Preprocessing** transformation of the instance matrix so that every node starts with 0
- **Constraints** these are similar between all three models, although in the development some were changed, deleted or altered to tackle specific problems.

1.2 Docker

In order to make our project easily reproducible, we have implemented the use of Docker container. We ensure the creation of a consistent environment that can be deployed across different host systems, by including all the necessary solvers and dependencies in the Docker image.

2 Constraint Programming (CP) Model for MCP Problem

The MCP problem can be effectively modeled using Constraint Programming (CP). The key idea is to define a set of constraints that represent the problem's

requirements and then use a solver to find the optimal or feasible solution that satisfies all the constraints. Below, we describe the CP model for the MCP problem using mathematical notation.

2.1 Decision variables

Let M be the number of couriers and N be the number of items to be delivered. We define the following variables:

- M: The total number of couriers.
- N: The total number of items to be delivered.
- l_i: The maximum load capacity for each courier i.
- s_j: The weight of each item j.
- D: The distance matrix between all pairs of locations (items and base).
- $order\_of\_delivery_{ij}$: The j-th location visited by courier i.
- $maximum\_distance$: The maximum distance traveled by any courier.
- $courier\_distances_i$: The total distance traveled by courier i.

2.2 Objective Function

The objective is to minimize the maximum distance traveled by any courier:

$$\text{Minimize } maximum\_distance \tag{1}$$

This formulation provides a comprehensive model for the MCP problem using Constraint Programming (CP), which can be solved using solvers such as Gecode or Chuffed.

2.3 Constraints

The following constraints are defined to model the MCP problem:

1. **All items must be delivered by exactly one courier:**

$$\forall j \in \{1, \ldots, N\}, \exists i \in \{1, \ldots, M\}, \exists k \in \{2, \ldots, N{+}1\} : order\_of\_delivery_{ik} = j \tag{2}$$

2. **The total weight carried by each courier must not exceed their capacity:**

$$\forall i \in \{1, \ldots, M\}, \sum_{k=2}^{N+1} s_{order\_of\_delivery_{ik}} \leq l_i \tag{3}$$

3. **Each courier starts and ends their route at the base:**

$$\forall i \in \{1,\ldots,M\}, order\_of\_delivery_{i1} = N{+}1 \wedge order\_of\_delivery_{i(N+2)} = N{+}1 \tag{4}$$

4. **Couriers should start with a delivery right after leaving the base:**

$$\forall i \in \{1,\ldots,M\}, \exists k \in \{2,\ldots,N{+}1\} : order\_of\_delivery_{ik} \neq N{+}1 \Rightarrow order\_of\_delivery_{i2} \neq N{+}1 \tag{5}$$

5. **Optional: If a courier is capable of carrying any item, they must carry at least one:**

$$\forall i \in \{1,\ldots,M\}, \exists j \in \{1,\ldots,N\}, \text{if } l_i \geq \max(s_j) \Rightarrow \exists k \in \{2,\ldots,N{+}1\} : order\_of\_delivery_{ik} \neq N{+}1 \tag{6}$$

6. **Minimize the maximum distance traveled by any courier:**

$$maximum\_distance = \max_{i \in \{1,\ldots,M\}} \sum_{k=1}^{N+1} D_{order\_of\_delivery_{ik}, order\_of\_delivery_{i(k+1)}} \tag{7}$$

2.4 Symmetry Breaking Constraints

To reduce the search space and improve solver efficiency, symmetry-breaking constraints are added:

1. **If two couriers have the same capacity, enforce a lexicographical order on their deliveries:**

$$\forall i, j \in \{1,\ldots,M\}, i < j \wedge l_i = l_j \Rightarrow order\_of\_delivery_i \leq_{lex} order\_of\_delivery_j \tag{8}$$

2. **If two couriers have similar routes, enforce a lexicographical order to break symmetry:**

$$\forall i, j \in \{1,\ldots,M\}, i < j \wedge \max(courier\_weight_i, courier\_weight_j) \leq \min(l_i, l_j) \Rightarrow order\_of\_delivery_i \leq_{lex} \tag{9}$$

2.5 Validation

In order to solve the MCP problem using the Constraint Programming (CP) model formulated in MiniZinc, we utilized a Python program to handle the data processing, solution execution, and result analysis. The program was designed to run two primary CP solvers: Gecode and Chuffed. The results were parsed and saved into JSON files for further analysis.

2.5.1 Python Program Overview

The Python program reads input data from text files, processes it into a format that MiniZinc can understand, and then executes the CP model using the specified solver. The results from each solver are then captured and processed to determine the effectiveness of the solution, particularly in terms of the objective function, which seeks to minimize the maximum distance traveled by any courier.

Key steps in the Python program include:

1. **Reading Input Data:** The program reads the number of couriers, the number of items to be delivered, their respective weights, and the distance matrix from a text file. The input data is then processed into arrays that match the structure expected by the MiniZinc model.

2. **Writing the .dzn File:** The input data is formatted into a '.dzn' file, which is the data format used by MiniZinc to provide instance-specific data to the model.

3. **Running the Solvers:** The program executes the MiniZinc model using both the Gecode and Chuffed solvers. It ensures that the solvers are allowed a maximum runtime (e.g., 300 seconds), after which the program forcefully terminates the solver if it hasn't finished.

4. **Parsing the Results:** After execution, the program parses the output from MiniZinc, extracting the solution, the maximum distance ('maximum distance'), and whether the solution was optimal.

5. **Saving Results:** The results are then saved in a structured JSON format, which includes the time taken by each solver, the sequence of deliveries, and the value of the objective function.

2.5.2 Results and Observations

- The Gecode solver generally provided quicker results than Chuffed in smaller instances, but but Chuffed sometimes found better solutions when allowed to run longer.

- Both solvers struggled with very large instances, where timeouts often occurred. This highlights the importance of choosing the right solver based on the specific problem instance.

- The symmetry-breaking constraints proved essential in reducing the search space and improving solver efficiency, especially in cases with identical or nearly identical couriers.

2.5.3 JSON Output

The JSON files generated by the program include details such as the time taken by each solver, the optimized delivery routes, and the maximum distance traveled. The structure of these files makes it easy to analyze and compare results across different solvers and problem instances.

Instances	Gecode	Chuffed
inst01	14	14
inst02	226	226
inst03	12	12
inst04	220	220
inst05	206	206
inst06	322	322
inst07	Inf	Inf
inst08	186	186

Table 1: Results for Gecode and Chuffed for each instance
N.B! Instances 9-21 all time-out; that is why they are not included in the table

3 SMT

SMT, or satisfiability modulo theory, concerns the study of satisfiability in formulas taking into account some background theories. In order to assert the satisfiability there is the need to use specific solvers, called SMT solvers, and in this case we used Z3. The option was to use, for this project, either SAT or SMT, but ultimately we decided to use SMT as it extends SAT but also retains some interesting characteristics from CP, like domain-specific reasoning. It is also able to handle things like distances and loads, which are essential in a task like the one at hand.

3.1 Decision Variables

The following decision variables are defined in our SMT model:

- `x[i][j][k]`: **Sort:** `Bool`
  This Boolean variable is `True` if courier k travels from node i to node j, and `False` otherwise. It is used to specify if courier k decides to take a route or not, and it is useful as the numerical value of True is 1 and the numerical value of False is 0. The theory used is **Boolean Logic**.

- `courier_loads[k]`: **Sort:** `Int`
  This integer variable represents the total load carried by courier k.The theory used is **Linear Arithmetic**.

- `u[j]`: **Sort:** `Int`
  An integer variable used in the MZT approach to enforce ordering of the nodes in a courier's route, preventing subtours. In this specific case, the actual meaning of the u decision variable will be further explained in the section dedicated to the MZT constraints, but in short it represents the

position or weight of each node in the graph. The theory used is **Linear Arithmetic**.

- `objective`: **Sort:** `Int`
  This integer variable represents the maximum distance traveled by any courier. The solver attempts to minimize this value while satisfying all constraints. The theory used is **Linear Arithmetic**.

3.2 Objective function

The objective of this SMT model is to minimize the maximum distance traveled by any courier. This is particularly important in logistics optimization, where the goal is to ensure that no courier has an excessively long route compared to others, thereby balancing the workload.

Let `n_couriers` be the number of couriers and `G.edges` be the set of all possible routes between nodes in the graph. The distance between two nodes i and j is given by `distances[i][j]`. The decision variable `x[i][j][k]` is a Boolean that is `True` if courier k travels from node i to node j, and `False` otherwise.

3.2.1 Maximum distance

The maximum distance traveled by any courier is determined by iterating over all couriers and updating the maximum distance. This needs to be done because, unlike in MIP, we do not have a max() function max() which would calculate this value automatically. In this context, `max_distance` is initialized with the total distance of the first courier and is subsequently updated to reflect the maximum distance across all couriers.

3.2.2 Optimization Objective

The SMT solver is then tasked with minimizing the `max_distance`, which represents the maximum distance any courier travels. This is formalized by introducing an integer decision variable `objective` that is constrained to be equal to `max_distance`. It also imposes that the max distance needs to be greater than a lower bound calculated as the maximum roundtrip distance from the depot to a node. If an upper bound is provided, the solver also ensures that the objective does not exceed this upper bound. Thus, the SMT solver searches for a solution where the maximum distance traveled by any courier is minimized, ensuring an optimal and balanced distribution of routes among couriers. This is done through the find optimal function which searches for solutions until the best one is found or the time is up.

3.3 Constraints

- There is no route from a node to itself, meaning that x from i to i for any courier is false

- Every item must be delivered, meaning that each 3-dimensional column must contain only one true value.
- Every node should be entered and left once and by the same vehicle, so the number of times a courier enters a node should be the same as the number of times it leaves the node. This is done by imposing that the sum of the times a courier leaves a node is the same as the number of times a courier enters it. This should be true for every node and for every courier.
- Each courier leaves and enters exactly once in the depot:
  1. the number of predecessors of the depot is 1
  2. the number of successors of the depot is 1
- For each vehicle, the total load over its route must be smaller than its max load size
  1. the sum of the package sizes assigned to a courier equals the total load that courier carries.
  2. the courier load for each courier is not negative
  3. the courier load for each courier doesn't exceed its max load possible
- **symmetry breaking**: If courier A takes route 1 and courier B takes route 2, the solution where these routes are swapped is not explored to reduce the search space. This is implemented thanks to the MZT decision variable u which imposes an order of the couriers themselves.
- **implied**: All the other nodes to be visited come after the depot and the total load of all vehicles doesn't exceed the sum of vehicles capacities
- **MTZ**:These constraints ensure the absence of subtours in the solutions. This ensures that each courier completes a loop from and to the depot, but it doesn't compute any loops around the track. These constraints were proved to be essential for the correct functioning of the model. These are based on assigning a u value to each node in the graph. These are in common between SMT and MIP.
  1. set the weight (or position) of the depot node, which in this case is defined as 1
  2. the weight of all other nodes (no depot) is at least 2
  3. the only accepted travel is to a node of higher weight

3.4 Validation

In this section I will illustrate how the validation step works. The solver used is z3 **Solver**. Some tools that aid the creation of a solution are:

- The use of a lower bound which helps reduce drastically the search space. This is calculated as the round trip distance between the depot and the furthest node in the graph.
- The use of an upper bound which is dynamically defined and updated. This upper bound starts out as *None* in the first search, and if a solution is produced within the given time, then another search starts which in this case has the time left and the newly found objective function as parameters. This search is conducted in the find optimal function, and it carries on until the optimal solution is found or the time limit is reached.
- creation of a python script that runs the model in order to set an external timeout, as it was necessary to ensure that it was respected.

Furthermore, the code runs 4 configurations, which are:

1. Default, no implied constraints and no symmetry breaking
2. With implied constraints
3. With symmetry breaking
4. With both implied and symmetry breaking

In the following table the results are shown for each configuration:

Instances	Configuration 1	Configuration 2	Configuration 3	Configuration 4
inst01	14	14	14	14
inst02	226	226	226	226
inst03	12	12	12	12
inst04	220	220	220	220
inst05	206	206	206	206
inst06	322	322	322	322
inst07	303	279	307	312
inst08	186	186	186	186
inst09	436	436	436	436
inst10	244	244	244	244

Table 2: Results for each configuration for each instance

The instances from 11 to 21 are all deemed unfeasible by the solver.

4 MIP Model

The third method we have used for searching a solution for Multiple Couriers Planning (MCP) problem is MIP. MIP models consist of a combination of integer and continuous variables that are optimized according to a linear objective function, subject to a set of linear constraints. Solvers like Gurobi are equipped with algorithms that efficiently explore the solution space, to find optimal or near-optimal solutions within a reasonable time frame.

4.1 Decision variables

Decision Variables

- **Binary Variable** $x[k, i, j]$:
  - **Domain**: $x[k, i, j] \in \{0, 1\}$
  - **Description**: The variable $x[k, i, j]$ indicates whether courier k travels directly from point i to point j. Specifically:
    - $x[k, i, j] = 1$ if courier k travels from point i to point j.
    - $x[k, i, j] = 0$ otherwise.
  - **Indices**:
    - k: Index for couriers, where $k = 1, \ldots, m$.
    - i, j: Indices for points, where $i, j = 1, \ldots, n + 1$.
- **Integer Variable** $u[i]$:
  - **Domain**: $u[i] \in \{1, \ldots, n\}$
  - **Description**: The variable $u[i]$ represents the sequence in which the points are visited by a courier. For each point i:
    - $u[i] = j$ implies that point i is the j-th point visited in the sequence by the courier.
  - **Indices**:
    - i: Index for points, where $i = 1, \ldots, n$.

4.2 Objective function

In this model, the objective variable `maxTravelled` is introduced to represent the maximum distance traveled by any courier. This variable is indeed crucial for our goal, which is to minimize the maximum distance traveled by any courier, ensuring a fair and balanced workload distribution among the couriers. The variable `maxTravelled` has a lower bound, which is calculated as the maximum round-trip distance from the origin to any distribution point and back. This lower bound ensures that `maxTravelled` cannot be smaller than the minimum possible longest route, which is the farthest distance any courier would have to travel to service a single point. The lower bound is computed as follows:

$$\texttt{lower\_bound} = \max\left(\texttt{all\_distances}[0, i] + \texttt{all\_distances}[i, 0]\right) \quad \forall i \in G.\text{nodes}$$

where G.nodes represents all nodes in the graph, and $\texttt{all\_distances}[0, i] + \texttt{all\_distances}[i, 0]$ calculates the round-trip distance from the origin to any node i and back. The objective function is formulated as:

```
minimize  maxTravelled
```

subject to the constraint:

$$\sum_{(i,j)\in\text{edges}} \texttt{all\_distances}[i,j]\times x[k][i,j] \leq \texttt{maxTravelled}, \quad \forall k \in \{1,\ldots,n\_couriers\}$$

This constraint ensures that the distance traveled by each courier k does not exceed the value of `maxTravelled`. The goal is to minimize this maximum distance, which effectively balances the load by minimizing the longest individual courier route. The objective function, therefore, seeks to minimize `maxTravelled`, ensuring that the maximum distance traveled by any courier is as small as possible.

4.3 Constraints

4.3.1 Default Constraints

1. **Item Assignment Constraint**: Each item must be assigned to exactly one courier, and each courier must deliver at least one item.

$$\sum_{k=1}^{m} \sum_{i\in G.nodes} x[k][i,j] = 1 \quad \forall j \in G.nodes,\ j \neq 0 \tag{10}$$

   Explanation: This constraint ensures that each item j is delivered by exactly one courier, ensuring complete coverage of all items.

2. **Vehicle Routing Constraints**: Each courier must enter and leave each node exactly once, maintaining flow conservation throughout the route.

$$\sum_{j\in G.nodes, j\neq i} (x[k][i,j] - x[k][j,i]) = 0 \quad \forall i \in G.nodes, \forall k \in \{1,\ldots,m\} \tag{11}$$

   Explanation: This ensures that for each courier k, the number of times they enter a node i is equal to the number of times they leave it, thereby preventing any vehicle from getting stuck.

3. **Depot Entry and Exit**: Each courier must start and end their route at the depot.

$$\sum_{i\in G.nodes, i\neq 0} x[k][i,0] = 1 \quad \text{and} \quad \sum_{j\in G.nodes, j\neq 0} x[k][0,j] = 1 \quad \forall k \in \{1,\ldots,m\} \tag{12}$$

   Explanation: This constraint ensures that each courier begins and ends their route at the depot.

4. **Load Constraints**: The total load carried by each courier must not exceed their maximum load capacity.

$$\sum_{(i,j)\in G.edges} size\_item[j] \times x[k][i,j] \leq max\_load[k] \quad \forall k \in \{1,\ldots,m\} \tag{13}$$

Explanation: This constraint ensures that no courier exceeds their maximum load capacity, maintaining feasibility in terms of vehicle load.

4.3.2 Implied Constraints

1. **Visiting Constraint**: All nodes must be visited after the depot, enforcing a logical sequence in the routing.

$$u[i] \geq 2 \quad \forall i \in G.nodes,\ i \neq 0 \text{ if configuration is } \{2,4\} \tag{14}$$

Explanation: This constraint ensures that all nodes are visited in sequence after the depot.

2. **Total Load Limitation**: The total load across all couriers must not exceed the sum of all couriers' capacities.

$$\sum_{k=1}^{m} \sum_{(i,j)\in G.edges} size\_item[j] \times x[k][i,j] \leq \sum_{k=1}^{m} max\_load[k] \tag{15}$$

Explanation: This constraint is implied by the problem setup, ensuring that the total load planned does not exceed the overall system's capacity.

4.3.3 Symmetry Breaking Constraints

1. **Distance Comparison Between Couriers**: The total distance traveled by one courier should not exceed that traveled by the next courier.

$$\sum_{(i,j)\in G.edges} all\_distances[i,j] \times x[k][i,j] \leq \sum_{(i,j)\in G.edges} all\_distances[i,j] \times x[k+1][i,j] \quad \forall k \in \{1,\ldots,m-$$

(16)

Explanation: This constraint enforces a logical ordering in the distance traveled by couriers, reducing the symmetry in the solution space and speeding up the search process.

4.3.4 Sub-Tour Elimination Constraints

Purpose: Sub-tour elimination constraints ensure that no courier creates a sub-tour that does not include the depot.

1. **Miller-Tucker-Zemlin formulation**: See the SMT section.

2. **Flow-Based Sub-Tour Elimination** (For configuration 5): The flow conservation constraints ensure that the flow into a node matches the demand, accounting for flow out. At the depot, flow is set equal to the total items carried. Capacity constraints ensure flow on any edge doesn't exceed the courier's capacity and only occurs if the edge is used. These constraints together prevent invalid routes, such as subtours, by enforcing proper flow distribution.

4.4 Validation

Instance	Config 1	Config 2	Config 3	Config 4	Config 5
1	14.0	14.0	14.0	14.0	14.0
2	226.0	226.0	226.0	226.0	226.0
3	12.0	12.0	12.0	12.0	12.0
4	220.0	220.0	220.0	220.0	220.0
5	206.0	206.0	206.0	206.0	206.0
6	322.0	322.0	322.0	322.0	322.0
7	167.0	167.0	167.0	167.0	167.0
8	186.0	186.0	186.0	186.0	186.0
9	436.0	436.0	436.0	436.0	436.0
10	244.0	244.0	244.0	244.0	244.0
11	Inf	Inf	Inf	Inf	Inf
12	593.0	1302.0	Inf	Inf	1109.0
13	488.0	468.0	602.0	744.0	398.0
14	Inf	Inf	Inf	Inf	Inf
15	Inf	Inf	Inf	Inf	Inf
16	286.0	286.0	Inf	Inf	286.0
17	Inf	Inf	Inf	Inf	Inf
18	Inf	Inf	Inf	Inf	Inf
19	334.0	334.0	Inf	Inf	718.0
20	Inf	Inf	Inf	Inf	Inf
21	Inf	Inf	Inf	Inf	Inf

Table 3: 1: Default constraints, 2: Implied constraints, 3: Symmetry Breaking constraints, 4: All the constraints, 5: Flow-Based Sub-Tour Elimination

5 Conclusions

CP proved to be flexible and efficient for moderate-sized instances, while SMT was effective with more complex ones but struggled with the larger instances of the problem, struggling especially with enforcing the timeout. MIP performed well in most of the scenarios but encountered infeasibility in more complex instances. Looking at the tables we can see that the model that struggled the least even with more complex instances is MIP, while CP times out for instances that the other models are able to find easily. Of course, some things like the computational speed of the machine that is running the code also affect the results. In spite of the challenges, all models pass the solution checker and find the best possible solutions for the first instances.