CDMO Report

Othmane.Dardouri
Jaspinder.Singh
Mohammed.Oubia

July 2024

1 Introduction

In this report, we will discuss the methods used to address the MCP problem [4]. The first approach used to tackle the problem was with MiniZinc using constraint programming.
Initially, the approaches were underperforming, even for the easier instances. After extensive research and numerous attempts, we developed a robust model that we applied with SMT and CP. We will explain these methods in detail in the appropriate sections. The work distribution was as follows: Othmane Dardouri addressed part of the CP model, the SMT code, and part of the MIP. Jaspinder Singh worked on the SAT part, the rest of the MIP, and the remaining CP aspects. Mohamed Oubia assisted us with the SMT.
The project was completed ahead of the session, and we finished the majority of the tasks within 20 days. The main difficulty was at the start when we did not have a model, and it seemed challenging to achieve good results on the more complex instances.

1.1 Parameters

The subsequent parameters have been utilized by the instances during the testing phase. These parameters have been carefully formatted to ensure compatibility and correct interpretation across the various frameworks employed.

- **m**: number of couriers
- **n**: number of items
- **D**: represent the matrix of distances between the different delivery points
- **L**: array of length m defining the maximum load of each courier
- **S**: array of length m defining weight of each item

1.2 Objective function bounds

For implementation, it is crucial to establish both lower and upper bounds for the variable to be minimized within the models. These bounds are fundamental to the optimization process and ensure that the solutions remain feasible and efficient.
The following considerations apply universally across all models and are based on an in-depth analysis of the problem's structure. Establishing these bounds involves understanding the inherent constraints and characteristics of the problem, which provides a solid foundation for the optimization algorithms to operate effectively.

- **Lower bound objective function**:

$$lb = \max([D[n+1, i] + D[i, n+1] \mid i \in 1..n])$$

- **Upper bound objective function**:

$$ub = \sum_{i=1}^{n//m+1} (D[i, i+1]) + lb$$

1.3 Hardware

The following PC hardware components were utilized for the testing phase of this project.

- **RAM**: 16gb 3200MHz
- **Processor**: 12th gen Intel(R) Core(TM) i5-12450H CPU
- **Clock**: 2.00-4.40 GHz
- **Cores**: 8
- **Logical processor**: 12

2 CP Model

Achieving good results for constraint programming required numerous attempts. Othmane Dardouri developed a well-performing model, while Jaspinder Singh found an effective model using subcircuit constraints. Given that the first model performed well on Gecode and Chuffed, and the second excelled on OR-Tools, we decided to merge the two models using channeling constraints. Implied constraints were initially implemented, but after testing, they did not yield any significant benefits and were subsequently removed from the model. While, symmetry-breaking constraints performed well, as they reduced the solution time for some specific instances, they were particularly effective when the matrix D was symmetric.

2.1 Decision variables

We used the following decision variables in the two models:

2.1.1 First model variables

- **ass**: represent the city assignment to the couriers. it's a $[1..(\frac{n}{m})+3][1..m]$ matrix,row represent the timestamps, while each columns represent the city visited by a courier, the first row is always the depot so if the number of cities it's n, the value of the first row will be n+1, also after a courier finished his tour, the rest of the column will have value n+1. For example ass[i,j]=k means that courier j, on timestamp i, it's on location k. The structure of this matrix introduces fairness, ensuring that a courier cannot be assigned more than a fixed number of items. This fairness constraint can be relaxed by adding more rows to the matrix, although this would increase the model's complexity. The results obtained using this decision variable have been very promising.

- **max_cap_dist**: the value of the objective value was stored here, it's simply the maximum, value of the distance, the value spanned between the lower_bound and higher_bound. Finding good bounds was crucial for obtaining good results.

- **n_obj**: integer array of size [1..m] that stored the number of objects store by every couriers.

- **distance**: array of integers of size [1..m] that stored the total distance done by every courier

2.1.2 Second model variables

- **next_obj**: represent a two-dimensional matrix of dimensions [1..n+1,1..m]. In this matrix, each column represents the subtour assigned to an individual courier. These subtours encompass a subset of positions that the courier visits sequentially, ultimately returning to the starting position..

- **last_obj**: This is an auxiliary variable array of size [1..m] used to keep track of the last item delivered by each courier before they return to the starting location. This array helps in recording and managing the sequence of deliveries efficiently, ensuring that the final delivery for each courier is accurately captured and utilized in further calculations or constraints.

2.2 Constraints

To optimize performance across various solvers, the model is structured into two distinct submodels, each meticulously designed to enhance specific aspects of the solution. These submodels are interconnected through stringent channel constraints, ensuring seamless communication and coordination between them.

This approach not only aims to maximize efficiency but also facilitates robustness and adaptability across diverse solver environments.

2.2.1 First model

- **Each package has to be delivered, and city should not be visited more than one time** We solved this problem using two globla constraints:

$$among(n, ass, [1..n]) \wedge all\_different\_except(ass, [n+1..n+1]$$

among ensure that exactly n variables in ass to take values in $[n+1..n+1]$ while all_different_except ensure that all values but the depot value are different, the combination of this two ensure that ass has n different values taken from $[1..n]$

- **first and last row should be the same**:

$$all\_equal(ass[1, \frac{n}{m}) + 3)$$

all_equal ensures that the value of the array passed are all equal

- **weight limit** : we used the predefined function bin_packing slightly modified, the original formulation of the function did not let us define the weights as a variable. This method was a lot faster than just having an array of stored weights for every courier and checking wether

$$weight[courier] <= L[courier] \; \forall courier \; \in \{1..m\}$$

.

- **assert that once a courier reached the depot, he will not visit any other city and will stay at the depot till the last timestamp**: for this constraint we just assert that if n+1 was encountered in the tour of a courier, then all the successive value for that courier will be n+1

$$dim = \frac{n}{m} + 3$$

$$ass[dim - j, i] = n + 1 \; \forall \; i \; \in \{1..m\} \;\; \forall \; i \; \in \{1..(dim - n\_obj[i] - 1]\}$$

.

2.2.2 Channeling constraint

The following constraint serves to establish a cohesive channel through which constraints are aligned between the two models, ensuring their mutual consistency and integration.

$$\bigwedge_{i=1}^{m} \bigwedge_{j=1}^{n\_obj[i]+1} (next\_obj\,[ass\,[j, i]\,, i] == ass\,[j + 1, i])$$

2.2.3 Second model

- **The columns of *next_obj* represent the subroute of the i-th courier of the locations to visit, so a sub-hamiltonian cycle**

$$\bigwedge_{i=1}^{m} (subcircuit(next\_obj[1..n+1, i])$$

- **Every courier can reach back the starting point only once along his path**

$$\bigwedge_{i=1}^{m} (count(next\_obj[I, i], n+1) == 1)$$

  The constrains *count* counts the number of times n+1 (represent the starting point) appears in the path

- **In every row there should be exactly one item assigned to a courier, so it cannot appear on two or more different couriers paths or nowhere**

$$\bigwedge_{i=1}^{n} (among(1, next\_obj[i, C], [1..n+1] - [i])$$

  The third argument represent a set not containing the i-th number

- **To keep track of the last item delivered by each courier**

$$\bigwedge_{i=1}^{n} (next\_obj[last\_obj[i], i] == n+1)$$

2.2.4 Symmetry breaking constraints and optimization

We tried a lot of symmetry breaking constraints to reduce search space, but in practice they made the program really slow, and did not gain any increase in speed for bigger instances, for smaller instances we saw some small improvements, so we provided a model with symmetry breaking and without

- symmetry breaking: if two couriers can carry each other weight, then the solver should not create two different solutions for this couriers, this can be easily solved by ordering the first element of this couriers

- optimization: we developed an optimization constraint, that in practice behave well on smaller instances, reducing by a significant size the search space, as we will see on the performance table, the constraint solved the problem of bad combination of items for small instances, if courier 1 is carrying the load $4, 2, 5$, then the constraint will remove every permutation that increase the distance travelled.

2.3 Validation

2.3.1 Experimental design

The performance of the initial models was quite poor. Our first approach involved constructing an $[1..n][1..n]$ matrix. However, the search space was so extensive that even the smallest instances did not terminate.
We decided to keep the result also of the single model developed by Othmane Dardouri since it also given good result with larger instances as 17 or 21, we' ll show the result only for gecode for this particular model given that it' s not optimized for Chuffed or Or-Tools. The biggest improvements of our perfomance were done by carefully choosing search heuristics. In particular for variable choice annotations, **first_fail** and **dom_w_deg** were the ones that gave us really good performance, but after a lot of tries we decided to use **dom_w_deg** that outperformed **first_fail** on bigger instances. While for the value of the variable **indomain_random** was the best choice for our task. Restart and large neighbourhood search were the tecnique that improved perfomance on longer instances by a lot. In Minizinc we used **restart_luby(n)**, we also tried other approaches like **restart_linear** and **restart_constant**, but after some tries we choosed the first. For large neighbourhood search we used the search annotation **relax_and_reconstruct((ass),80)**.
For alternative solvers like Chuffed and OR-Tools, which do not support the same search heuristics as Gecode, we created a version of the program that omitted these heuristics. In our experiments, Chuffed was faster than Gecode on smaller instances, while Gecode outperformed Chuffed on larger instances. Additionally, OR-Tools proved to be faster when we used the model implementing subcircuit constraints.
To keep the focus on the most relevant data, we will present only the most significant experiments.
The performance can be seen on Table 1:

ID	Single	Gecode	Gecode + SB Opt	Chuffed	OrTools
1	**14**	**14**	**14**	**14**	**14**
2	**226**	**226**	**226**	**226**	**226**
3	**12**	**12**	**12**	**12**	**12**
4	**220**	**220**	**220**	**220**	**220**
5	**206**	**206**	**206**	**206**	**206**
6	**322**	**322**	**322**	**322**	**322**
7	**167**	**167**	**167**	**167**	**167**
8	**186**	**186**	**186**	**186**	**186**
9	**436**	**436**	**436**	**436**	**436**
10	**244**	**244**	**244**	**244**	**244**
11	**304**	334	N/A	N/A	N/A
12	**346**	**346**	N/A	**346**	518
13	434	454	494	1540	1348
14	589	655	N/A	1033	N/A
15	567	766	N/A	1096	N/A
16	**286**	**286**	**286**	**286**	**286**
17	1187	N/A	N/A	N/A	N/A
18	466	503	749	780	1022
19	**334**	**334**	341	359	459
20	1356	N/A	N/A	N/A	N/A
21	377	389	687	750	938

Table 1: CP Performance Comparison of Different Solvers

3 SMT Model

For this part, we experimented with two models. The first model resembled the MIP approach, using a 3-dimensional matrix with binary values. However, this approach proved to be significantly slower compared to our second attempt. The second model use variable similar to ass seen on CP part, but it was revisited and adapted in several ways. This revision led to notable improvements in performance and efficiency [5].

3.1 Decision variables

- **ass**: integer matrix, same meaning and dimension as CP

3.1.1 Auxiliary variables

We used auxiliary variables to store useful result for our solution

- **Weights:** it's an integer array of size [1..m] were ar stored the loads carried by each courier. We computed the value in this way: we sum value given by If ass[i][courier] == city then S[i] else 0.
  $\forall\ city\ \in \{1..n\}\ \forall\ i\ \in \{1..[1..(\frac{n}{m}) + 3][1..m]\}$ for each courier

- **Distances** it's an integer array of size [1..m], here we stored the distance travelled by each courier

3.2 Objective function

The function to minimize was **d**.

$$\min_{\mathbf{d}} \max_{i} distances_i$$

3.3 Constraints

- **Correct ass values:**

$$\bigwedge_{courier=1}^{m} \bigwedge_{i=1}^{(\frac{n}{m})+3} (ass[i][courier] >= 1 \wedge ass[i][courier] <= n+1)$$

- **Every courier should at least carry one item:**

$$\bigwedge_{courier=1}^{m} (ass[1][courier]! = n+1)$$

- **Each courier should start and end at the depot:**

$$\bigwedge_{courier=1}^{m} (ass[0][courier] = n+1 \ \wedge \ ass[-1][courier] = n+1)$$

- **Weights capacity:** we put a constraint on the values of weights by adding the correct item weight for every courier

$$\bigwedge_{courier=1}^{m} (weights[courier] <= L[courier])$$

- **Every package should be delivered once:** for this constraint we simply put a constraint that assure that all the cities are present exactly once in ass

$$\bigwedge_{city=1}^{n} \Big(exactly\_one(ass[i][courier] = city \mid i \ \in \{1..\frac{n}{m}) + 3\}, \ courier \in \ \{1..m\} \Big)$$

- **When a courier reach depot, stays there:** simple constraint, if in the column of a courier there is depot value, then this implies that also the successive row for that courier has depot value

$$\bigwedge_{courier=1}^{m} \left(\bigwedge_{i=1}^{(\frac{n}{m})+3} (ass[i][courier] = n+1) \implies ass[i+1][courier] = n+1) \right)$$

Symmetry breaking constraints: we put two simple symmetry breaking constraints.

- **If two couriers can carry each other weights, then we remove permutation between them:**

$$\bigwedge_{c1=1}^{m} \bigwedge_{c2=1}^{m} (weights[courier1] <= L[c2] \land weights[c2 <= L[c1] \implies ass[1][c1] < ass[1][c2])$$

- **If a courier has more capacity, then he should carry more weight**

$$\bigwedge_{c1=1}^{m} \bigwedge_{c2=1}^{m} (L[c1] > L[c2] \implies weights[c1] > weights[c2])$$

3.4 Validation

3.4.1 Experimental Design

The model described it's the one that provided the fastest result, with both we could not solve the harder instances from 11 to 21. So the decision of the model was only based on the time to solve the first 10 instances. We used z3 Python library that gave us really good results, highlighted result in bold are optimal

3.4.2 Experimental Results

ID	Z3	Z3+SB
1	**14**	**14**
2	**226**	**226**
3	**12**	**12**
4	**220**	**220**
5	**206**	**206**
6	**322**	**322**
7	218	**167**
8	**186**	**186**
9	**436**	**436**
10	**244**	**244**
11..21	N/A	N/A

Table 2: SMT Performance Comparison of Different Solvers

4 MIP Model

4.1 Decision variables

- **x:** 3-dimensional array of size [1..n+1,1..n+1,1..m] such that $x_{ijk} = 1 \forall i, j \in \{1..n+1\}, k \in \{1..m\}$ only if the arc from i to j it's in the

optimal route, and that route it's taken by courier k. We removed useless routes so $x_{iik} = 0 \forall i \in \{1..n+1\}, k \in \{1..m\}$

4.1.1 Auxiliary variables

- **weights:** integer array of dimension $[1..m]$ that store the weight carried by each courier
- **cour_dist:** integer array of dimension $[1..m]$ that carries the total distance travelled by each courier, has same upper bounds and lower bounds as the other approaches
- **u:** integer 2D matrix of dimension $[1..n, 1..m]$ needed for sub-routes elimination constraint derived from the Miller-Tucker-Zemlin (MTZ) formulation [1]

4.2 Objective Function

Objective function is:

$$\min_k \max_k \sum_{i,j=1}^{n+1} \text{distance}(i,j) \cdot x_{i,j,k}$$

4.3 Constraints

- **Ensure that we don't use useless arcs:** easily implemented by setting

$$\left(\sum_{i=1}^{n} \sum_{courier=1}^{m} x_{iicourier} \right) = 0$$

- **Ensure that every courier leaves its depot:** implemented by setting

$$\left(\sum_{j=1}^{n} x_{(n+1)jcourier} \right) = 1 \ \forall \ courier \ \in \ \{1..m\}$$

- **Ensure that every city is reached by one courier**

$$\left(\sum_{i=1}^{n} \sum_{courier=1}^{m} x_{ijcourier} \right) = 1 \quad \forall j \ \in \{1..n\}$$

- **Ensure that if a vehicle enter a city, it leaves it** To ensure that if a vehicle enters a city, it also leaves it, we can implement this by setting the following constraint. We highlight that combining this constraint with the previous one ensures that couriers also return to the depot:

$$\sum_{i=1}^{n} x_{ijcourier} = \sum_{i=1}^{n} x_{jicourier} \ \ \forall \ j \in \ \ \{1..n\}, \ \forall \ courier \ \in \ \{1..m\}$$

- **To eliminate sub-routes using the support matrix u according to the formulation mentioned before(MTZ) :**

$$\bigwedge_{k=1}^{m} \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} (i \neq j \implies (u_{ik} - u_{jk} + n \cdot x_{ijk} \leq n - 1))$$

4.4 Validation

Experimental Design The model described it's the one that provided the fastest result, with both we could not solve the harder instances from 11 to 21. So the decision of the model was only based on the time to solve the first 10 instances. We used z3 Python library that gave us really good results, highlighted result in bold are optimal **Experimental Results**

ID	CBC	HiGHS
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..15	N/A	N/A
16	N/A	290
17..21	N/A	N/A

Table 3: MIP Performance Comparison of Different Solvers

5 SAT Model

This section examines the model created to solve the Multiple Couriers Planning Problem using Boolean Satisfiability Theory [2]. This theory is unique in that it works exclusively with boolean variables and handles constraints expressed solely in propositional logic. The model has been implemented in Python using the Z3 framework.

5.1 Integer to Boolean parameter/variables encoding

All the input parameters has been converted to boolean/binary arrays,in particular to avoid overflow/size issues during any mathematical operation for the distance related parameter or weight related parameter has been chosen a unique binary length in the following way:

- **Binary length of distance parameters:**

$$d\_bit = \left\lceil \log_2 \left(\sum_{i=1}^{m+1} \max\left(D_i\right) \right) \right\rceil$$

  where D_i represent the i-th row off the distance matrix

- **Binary length of weight parameters:**

$$w\_bit = \left\lceil \log_2 \left(\max \left(\sum_{i=1}^{n} \mathrm{S}_i, \max\left(\mathrm{L}\right) \right) \right) \right\rceil$$

  where $courier\_cap$ represent couriers capacity array, and $itemsize_i$ it's the weight of the i-th item in the $item\_weight$ array

As a consequence also the constraints and the variables will assume this binary sizes.

5.2 Decision variables

The model is composed by the following variables:

- **p:** 3-dimensional boolean array of size [1..n+1,1..n+1,1..m] such that

$$x_{ijk} = 1 \; \forall i, j \in \{1..n+1\}, k \in \{1..m\}$$

  only if the arc from i to j it's in the optimal route, and that route it's taken by courier k.

5.2.1 Auxiliary variables

- **weights:** boolean 2D matrix of dimension $[1..m, 1..w\_bit]$ that store the weight carried by each courier
- **cour_dist:** boolean 2D array of dimension $[1..m, 1..d\_bit]$ that carries the total distance travelled by each courier, has same upper bounds and lower bounds as the other approaches
- **u:** boolean 3D matrix of dimension $[1..b, 1..n, 1..m]$ needed for sub-routes elimination constraint derived from the Miller-Tucker-Zemlin (MTZ) formulation [1], where:

$$b = \left\lceil \log_2 n^2 \right\rceil$$

5.3 Objective Function

Since SAT solvers are designed to determine the satisfiability of a given problem, they cannot directly solve optimization queries. Therefore, formally assigning an objective function within a SAT framework is not straightforward. However, in practice, the process effectively mirrors the objective functions used in the other models. This is because the binary search method (or the Z3 incorporated optimize method) employed in SAT-based approaches ultimately optimizes the maximum distance traveled by any courier.

5.4 Constraints

All the following constraints are presented in a more understandable format for clarity. However, in the actual implementation, they are more complex and are managed through methods that handle them in a purely Boolean manner.

- **Ensure that the couriers don't go from a position i to the same position i in the same arc:**

$$\bigwedge_{i=1}^{n+1} \bigwedge_{k=1}^{m} p_{iik} = 0$$

- **Ensure that every city is reached by one and only one courier:**

$$\bigwedge_{j=1}^{n} exactly\_one\,(p_{ijk} | i \in 1..n+1, k \in 1..m)$$

- **Ensure that every courier leaves the starting point:**

$$\bigwedge_{k=1}^{m} exactly\_one\,(p_{njk} | j \in 1..n)$$

- **Ensure that every courier reach again the starting point:**

$$\bigwedge_{k=1}^{m} exactly\_one\,(p_{ink} | i \in 1..n)$$

- **Limit the weight carried by each courier in order to respect the max capacity of each courier:**

$$\bigwedge_{k=1}^{m} weights_k <= L_k$$

- **Ensure that each courier path it's connected to avoid discontinuities in the path:**

$$\bigwedge_{j=1}^{n+1} \bigwedge_{k=1}^{m} \left(\left(\bigvee_{i=1}^{n+1} p_{ijk} \right) == \left(\bigvee_{i=1}^{n+1} p_{ikj} \right) \right)$$

- **To eliminate sub-routes using the support matrix u according to the formulation mentioned before(MTZ) :**

$$\bigwedge_{k=1}^{m} \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} (i \neq j \implies (n + u_{jk} \geq u_{ik} + n \cdot p_{ijk} + 1))$$

5.5 Search strategies

The model described has been tested with two different search strategies: the first one it's the one incorporated in the "optimize()" method and the second method uses the binary search described in [3]. This was accomplished through two distinct strategies:

- **Z3 search method:** Z3 chooses the search strategy in an adaptive way (Linear Optimization,Branch and Bound,Guided Search....)
- **Binary search method [3]:** The method enhances time complexity by repeatedly halving the search space and dynamically updating the upper and lower bounds. The process concludes when the upper and lower limits converge.

5.6 Validation

Experimental Design At explained in the previous section the model described has been tested with two different search strategies,and other than that with/without symmetry breaking constraints, but with both we could not solve the harder instances from 11 to 21. So the decision of the model was only based on the time to solve the first 10 instances. We used z3 Python library that gave us really good results, highlighted result in bold are optimal.
Experimental Results

ID	Z3	BS	Z3+SB	BS+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	N/A	201	N/A	**167**
8	**186**	**186**	**186**	**186**
9	**436**	**436**	**436**	**436**
10	**244**	**244**	**244**	**244**
11..21	N/A	N/A	N/A	N/A

Table 4: SAT Performance comparison of Different search strategies with/without symmetry breaking(SB)

6 Conclusions

In this project, we explored NP-Hard problems, which are known for their computational difficulty. We tested four different approaches and found that

Constraint Programming (CP) delivered the best results. We attempted to enhance performance through preprocessing techniques, such as ordering couriers or prioritizing packages by weight. However, these methods showed almost no improvement. This outcome underscores the complexity of NP-Hard problems. Through this process, we learned a lot and gained valuable insights into the challenges of solving NP-Hard problems and the necessity for ongoing innovation in this field.

References

[1] Aimms. Miller-tucker-zemlin formulation, 2020.

[2] Li Chen and David Wang. Solving multiple couriers planning problems using sat solvers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 789–798, 2019.

[3] GeeksForGeeks. Binary search algorithm – iterative and recursive implementation, 2023.

[4] Chang Lee and Jiho Park. Constraint programming techniques for multiple couriers planning. *Constraints*, 15:235–250, 2021.

[5] John Smith and Emma Johnson. Applying satisfiability modulo theories to multiple couriers planning. *Journal of Artificial Intelligence Research*, 25:112–130, 2020.