CDMO Project
by Optimization Wizard

Alessandro Pasi, alessandro.pasi8@studio.unibo.it
Alessio Pellegrino, alessio.pellegrino@studio.unibo.it
Lorenzo Massa, lorenzo.massa6@studio.unibo.it
Riccardo Murgia, riccardo.murgia2@studio.unibo.it

September 16, 2023

1 Introduction

This report addresses the Multiple Couriers Planning (MCP) problem, which is well-known to be NP-hard. We will outline some approaches based on Operation research strategies and declarative programming. The approaches involved are: Constraint Programming (CP), propositional SATisfiability (SAT), its extension to Satisfiability Modulo Theories (SMT), and Mixed-Integer linear Programming (MIP).
Our objective is to develop efficient and 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

The group work was essential for the success of this project. All members of the group actively participated in the problem modelling, implementation, testing, and documentation phases. In the early stages, we held several meetings to discuss how to model the problem finding a first approach. Subsequently, we found a new formulation and began implementing it. Alessio and Alessandro focused mainly on CP and SAT, while Riccardo and Lorenzo implemented SMT and MIP. Finally, Alessio and Riccardo refactored the code to ensure that all models were consistent, and Lorenzo and Alessandro performed testing and validation of the models. During the working period, a common text group and a weekly meeting helped us to keep up to date with each other's work and helped us with any possible doubts and problems. At first, the main difficulty was understanding how to model the problem. Once we found a first approach, we continued to research the topic by reading papers and articles. This led us to find a second model that in most of the cases offered better performance, as shown in the dedicated sections.

1.2 Benchmark setup

All the benchmarks were realized using a machine equipped with a Ryzen 1700 processor, 8GB of ram running Linux on kernel 6.4.6

1.3 Input parameters

This section is divided into two categories of parameters crucial for solving the Multiple Couriers Planning (MCP) problem: instance parameters that define the problem instance and adding parameters that improve the optimization process. A clear understanding of both types of parameters is essential to achieve optimal and efficient solutions.

1.3.1 Instance parameters

Instance Parameters encapsulate the inherent characteristics of the problem instance. We define the problem's context and properties:

- Number of Couriers (m): total count of couriers available for distributing items.
- Number of Items (n): quantity of items to be distributed $n > m$.
- Maximum Load Capacities (max_load): an array indicating the maximum load capacity of each courier
- Item Sizes (size): a list enumerating the sizes of individual items.
- Distance Matrix (distances): this matrix encapsulates the distances between customer locations.

1.3.2 Added parameters

These parameters are thoughtfully introduced by us to refine the optimization process: in particular, two upper bounds and two lower bounds have been defined to restrict the search space as much as possible. Let's define the lower bounds:

- Minimum number of packs (min_packs): This parameter defines the minimum number of packs a courier must deliver. Computed as:

$$\text{min\_packs} = \max(1, f(\sum_{i=1}^{n} size_i, loads, \{1, \ldots, n\}))$$

$$loads = \sum_{ls \in \text{max\_loads} \setminus \{min\_load\}} ls$$

$$min\_load = \min_{lds \in \text{max\_loads}} lds$$

Where:

$$f(l, m, ps) \begin{cases} 1 + f(l - size_p, m, ps \setminus \{p\})\ p = \underset{p \in ps}{\arg\max}\, size_p & \text{if } l > m \\ 0 & \text{if } l \leq m \end{cases}$$

And $size_p$ is the size of the pack p.

- Minimum Path Length (min_path): This parameter sets a lower bound on feasible path lengths. This value is computed starting from the min_packs value. In particular, we have two distinct cases: the case in which the min_packs value is 1 and the case in which the min_packs value is > 1. Let's analyze them independently:
  - min_packs = 1:

    $$\text{min\_path} = \max(\text{dist}_{o,i} + \text{dist}_{i,o})\ i \in \{1, \ldots, n\}$$

    Where $\text{dist}_{i,j}$ is the distance between the node i and j.
  - min_packs > 1: The following is not a precise solution since the distance between each node and the origin is different, however, we cannot compute a feasible min_path of length greater than 1 without an exhaustive search which will take a lot of computational time:

    $$\text{min\_path} = \max(\text{dist}_{o,i} + f(i, \{i\}, \text{min\_packs} - 1))\ i \in \{1, \ldots, n\}$$

    Where:

    $$f(x, set, s) \begin{cases} \text{dist}_{x,k} + f(k, set \cup \{k\}, s - 1),\ k = \underset{k \in \{1,\ldots,n\} \setminus set}{\arg\min}\, \text{dist}_{x,k} & \text{if } s \geq 1 \\ \min\limits_{\text{dist}_{k,o}} blue\text{dist}_{k,o},\ k \in \{1, \ldots, n\} & \text{if } s = 0 \end{cases}$$

    And $\text{dist}_{i,j}$ is the distance between node i and j.

Let's now define the upper bounds:

- Maximum number of packs (max_packs): This parameter defines the maximum number of packs a courier must deliver. Computed as:

  $$\text{max\_packs} = \min(n - m, f(0, \max_{l \in \text{max\_loads}} l, \{1, \ldots, n\}))$$

  Where:

  $$f(l, m, ps) \begin{cases} 1 + f(l + size_p, m, ps \setminus \{p\})\ p = \underset{p \in ps}{\arg\min}\, size_p & \text{if } l < m \\ 0 & \text{if } l \geq m \end{cases}$$

  And $size_p$ is the size of the pack p.

- Maximum Path Length (max_path): This parameter sets an upper bound on feasible path lengths. Computed as:

$$\text{max\_path} = \min_{i \in \{1,\dots,n\}} (dist_{o,i} + f(i, \{i\}, max\_packs - 1))$$

Where:

$$f(x, set, s) \begin{cases} \text{dist}_{x,k} + f(k, set \cup \{k\}, s-1),\ k = \underset{k \in \{1,\dots,n\} \setminus set}{\arg\max} \text{dist}_{x,k} & \text{if } s \geq 1 \\ \text{dist}_{x,o} & \text{if } s = 0 \end{cases}$$

And $\text{dist}_{i,j}$ is the distance between node i and j.

The main idea behind the min and max path is to enforce the triangle inequality of the graph described by the distance matrix. This, in conjunction with the knowledge about the number of packs, helped us develop feasible bounds for the distance.

1.4 Objective value

The goal of this instance of the Multiple Couriers Planning problem is to design an efficient distribution plan that minimizes the maximum distance traveled by any courier during the process of delivering items to various customer locations.

$$obj = \text{minimize} \max \sum_{i}^{o} \sum_{j}^{o} dist_{i,j} x_{i,j,k},\ k \in 1, \dots, m$$

Where $\text{dist}_{i,j}$ is the distance between node i and j and $x_{i,j,k}$ is a binary variable that is 1 if the courier k goes from i to j, 0 otherwise.

1.4.1 Experimental design

In this section, we are going to discuss the procedure we have followed to asses our models' performance. The procedure is the same for every model, in such a way we are able to compare them side by side. The hardware used is stated in section 1.2. The models were launched through docker. The docker image was built starting from the official minizinc docker image [8]. Only the CP model was executed with multithreading enabled (4 cores) while the other solvers were executed in a single thread. Each problem instance was then run 20 times to be sure the results were consistent (although the graph will show only the result of the last run with the purpose of showing an actual result). To reproduce our experiments it is necessary to:

- modify the *config.mcp* configuration file as shown in the *README* file in the repository
- build the docker image using the docker file in the repository

- execute the docker image

This will run each specified instance 1 time for each desired model and will save the result as a JSON file in the correct location in addition to showing the result in the standard output. We suggest, that the SAT, MIP and SMT models, to run only one instance at a time. This is because the Python garbage collector could, sometimes, fail to free the memory used for the previous instances, thus, for bigger instances, the memory could be saturated.

2 Constraint Programming

The constraint programming model is the one that differs the most from the others: although both for VRP [7] and TSP [3] problems the standard practice is to use the "circuit" constraint with a successor/predecessor model, after several empirical experiments, we found out that, in our cases, a model based of point in time, performed better.

2.1 Decision Variables

The principal decision variables utilized in the construction of this model encompass:

- *courier_route*: a 2D matrix of size $m \times (max\_packs+2)$ with integer values between 1 and n+1 used to define at each point in time, where courier j will go
- *courier_distance*: an integer array of size m with values between 1 and max_path. $courier\_distance_j$ contains the value of the distance the courier j will have to travel.
- *packs*: an array of size n with values between 1 and m. That contains the courier that must bring the package i (e.g. $packs_i = j$ means that courier j will have assigned pack i).
- *loads*: an array of size m with an integer value that contains the value of weight each courier must bring
- *max_distance*: the value of the objective function to minimize.

2.2 Constraints

The Constraint Programming (CP) model has been implemented employing a diverse set of constraints. Specifically, the following constraints have been introduced:

- a constraint that fixes the starting point of each courier $\forall j \in \{1, \ldots, m\} cr_{j,1} = o \vee cr_{i,max\_packs+1} = o$ with o being the origin and cr *courier_route*

- a constraint that forces each courier to have at least min packs $\forall j \in \{1,\ldots,m\}, i \in \{2,\ldots,min\_packs+1\} cr_{j,i} \neq o$ with o being the origin and cr *courier_route*
- bin_packing_capa: a global constraint that enforces that, the weight of each courier must not exceed his load capacity.
- a constraint that enforces that, $\forall j \in \{1,\ldots,m\}$ the value of $load_j$ to be equal to the sum of the packages assigned to courier j.
- a constraint that forces a courier to get back to the origin as soon as he (/she) has finished the deliveries.
  $\forall j \in \{1,\ldots,m\}, i \in \{2,\ldots,max\_packs+2\}$
  $cr_{j,i} = o \rightarrow \forall_{k=i}^{max\_packs+2} cr_{j,k} = o$ with o being the origin and cr *courier_route*
- a constraint that forces courier j to deliver pack i if it is assigned to him(/her) $\forall k \in \{1,\ldots,n\}, j \in \{1,\ldots,m\} packs_k = j \leftrightarrow \bigvee_i^{max_packs+2}$ $cr_{j,i} = k$ with cr being the *courier_route*. This constraint has been enforced using the member global constraint.
- a constraint that fixes at 1 the number of times a courier can be at any given place and fills the remaining free spot of the matrix with the origin. This constraint has been enforced using the global constraint global_cardinality_closed
- a constraint that forces the value of $distances_j$ to be the sum of the distances of each node traveled by j.
- Finally, a constraint that forces max_distance to be the maximum value of the distances has been added.

2.2.1 Implied Constraints

In addition to the previous constraint, it has been chosen to add an auxiliary constraint which speeds up the model. In particular, a constraint that enforces the fact that each courier must be at a different location from the others at any given moment (except if he(/she) is at the origin) has been added.

2.3 Symmetry breaking constraints

To break as many symmetries as possible, a symmetry-breaking constraint has been added. In particular, for each group of couriers with the same max_load, both the value of load and the value of distances must be increased in order to avoid two solutions with swapped values.

ID	G SB	G S	OT SB	OT S	C SB	C S
00	**12**	**12**	**12**	**12**	**12**	**12**
01	**14**	**14**	**14**	**14**	**14**	**14**
02	**226**	**226**	**226**	**226**	**226**	**226**
03	**12**	**12**	**12**	**12**	**12**	**12**
04	**220**	**220**	**220**	**220**	**220**	**220**
05	**206**	**206**	**206**	**206**	**206**	**206**
06	**322**	**322**	**322**	**322**	**322**	**322**
07	**167**	**167**	**167**	**167**	**167**	**167**
08	**186**	**186**	**186**	**186**	**186**	**186**
09	**436**	**436**	**436**	**436**	**436**	**436**
10	**244**	**244**	**244**	**244**	**244**	**244**
11-12	N/A	N/A	N/A	N/A	N/A	N/A
13	N/A	N/A	554	682	1786	2142
14-15	N/A	N/A	N/A	N/A	N/A	N/A
16	N/A	N/A	675	**286**	N/A	N/A
17-21	N/A	N/A	N/A	N/A	N/A	N/A

Table 1: Results for a given instance for solvers Gecode (G), Or-tools (OT), and Chuffed (C). SB stands for symmetry-breaking enabled while S stands for symmetry-breaking disabled

2.4 Validation

2.4.1 Experimental results

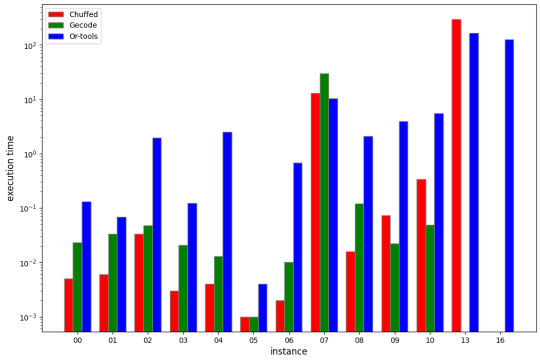


Figure 1: Time diagram for CP solvers with symmetry-breaking constraint enables

Between the tested solvers, Gecode is clearly the least performing one. Both Chuffed and Or-tools performed better with Or-tools being the most consistent one. However, even with multiple configurations, Or-tools keep crashing (for instance 13 and 16 it has never finished running even if there was still remaining time) with symmetry-breaking constraint enabled. The data is presented in logarithmic scale to improve readability.

3 SAT

3.1 Decision Variables

The principal decision variables utilized in the construction of this model encompass:

- *courier_route*: A z3.Bool matrix of shape $m \times (n+1) \times (n+1)$ used to indicate whether courier i traverses destination j.
- *load*: A z3.Bool matrix of shape $m \times binary\_lenght$ used to encode in binary values the load of each courier.
- *distance*: A z3.Bool matrix of shape $m \times binary\_lenght$ used to encode in binary values each distance.
- *max_distance*: A z3.Bool array of length *binary_lenght* used to indicate the maximum distance traveled by the couriers.
- *steps*: A z3.Bool matrix of shape $m \times n \times max\_packs$ used to encode the steps of each courier. This matrix is fundamental to avoid sub-tours.

3.2 Constraints

The SAT model has been implemented employing a diverse set of constraints. Specifically, constraints were introduced to guarantee that:

- Each courier starts and concludes deliveries at the origin node.(4)
- Each courier must stop in a position i if he has the pack i assigned, and then must move from there.
- Each courier is exclusively responsible for packages whose cumulative weight aligns with its carrying capacity.(3)
- Each position a courier visits must be in a moment greater than all the positions visited before.(5) (7)
- Every package must be conveyed to its designated destination, guaranteeing the location to be visited in a singular instance. (5)

The model, thus, has been modeled as:

$$\text{minimize} \max \sum_{i}^{o} \sum_{j}^{o} courier\_route_{k,i,j} d_{i,j} \quad k \in \{1, \ldots, m\} \tag{1}$$

$$\forall k \in \{1, \ldots, m\}, min\_packs \leq \sum_{i}^{n} courier\_load_{k,i} \leq max\_packs \tag{2}$$

$$\forall k \in \{1,\ldots,m\}, \sum_{i}^{n} courier\_load_{k,i} load_i \leq max\_load_k \tag{3}$$

$$\forall k \in \{1,\ldots,m\}, \bigvee_{i}^{n} courier\_route_{k,i,o} \wedge \bigvee_{i}^{n} courier\_route_{k,o,i} \tag{4}$$

$$\forall k \in \{1,\ldots,m\}, i \in \{1,\ldots,n\}, courier\_load_{k,i} \leftrightarrow \bigvee_{j}^{o} courier\_route_{k,i,j} \tag{5}$$

$$\forall k \in \{1,\ldots,m\}, i \in \{1,\ldots,n\} \bigvee_{j}^{n} courier\_route_{k,j,i} \rightarrow \bigvee_{f}^{o} courier\_route_{k,i,f} \tag{6}$$

$$\forall k \in \{1,\ldots,m\}, i,j \in \{1,\ldots,n\}, courier\_route_{k,i,j} \rightarrow steps_i < steps_j \tag{7}$$

3.2.1 Implied Constraints

In addition to the previous constraint, it has been chosen to add auxiliary constraints which speed up the model. In particular, The maximum distance traveled by couriers was limited using *min_path* as the lower bound and *max_path* as the upper bound so as to exclude impossible solutions from the search. Furthermore, a constraint that avoids loops of two nodes and a constraint that avoids a courier staying at a given location has been added

3.3 Number Encodings

The model needs to perform reasoning involving numbers to ensure that each courier carries an appropriate number of packs, facilitate distance computations, and enable comparisons of solution quality. To accomplish these objectives, two distinct encodings have been deployed:

- **Log Encoding:** This encoding is utilized for all variables in the model that necessitate summation with other numbers. The choice of binary encoding is motivated by its efficiency in terms of space, requiring only a logarithmic number of variables relative to the integer value, and its cost-effectiveness when used in operations. Only variables requiring computation by the solver, such as loads and distances, need to be transformed. Constant values like the distance between nodes remain unaltered. During summation, determining whether to add a number relies on a multiplier binary value (computed by the solver), resulting in a two-case function. This function, applied from least to most significant bit, facilitates binary Z3-encoded number addition with an integer. The function is defined as follows:

$$f(b,x,m,c) = \begin{cases} (\text{xor}(b,m,c), \text{Or}(\text{And}(b,c), \text{And}(b,m))) & \text{if } x = 1 \\ (\text{xor}(b,c), \text{And}(b,c)) & \text{if } x = 0 \end{cases}$$

A similar two-case function is used to check whether a Z3 boolean array represents a number in relation to a given integer value ($<, >, =, \geq, \leq$).

- **Order Encoding:** The order encoding of an integer n involves a boolean array with $k \geq n$ binary values. The i-th value is true only if $i = n$. Although this encoding is more space-intensive, it facilitates quicker checks. Consequently, it's specifically employed for subtour-elimination variables, where the summation is unnecessary and efficiency in checks is more important.

We also considered, as suggested by Bierlee, H et al.[4] to couple different encodings into our model. However, the number of operations needed to perform the conversions was much bigger than the number of saved operations we could have gotten by using only one of them.

3.3.1 Objective function

The objective function is the one described in (1). Since SAT methods focus only on satisfiability and not on optimization, to implement the search for an optimal solution we add a new constraint to the model every time a solution is found. The new constraint forces the objective function to be smaller than the one already found. To further speed up the search, we implemented a binary search algorithm.[5]

3.4 Validation

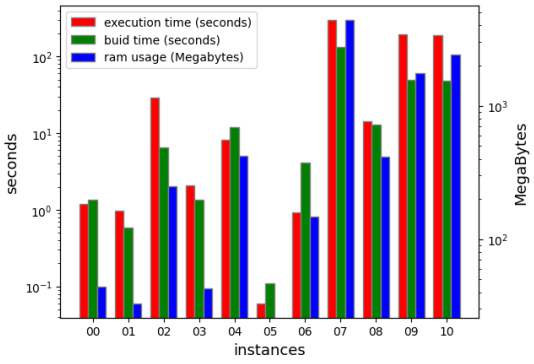


Figure 2: Time diagram for SAT solver

- An important aspect of the SAT model is the trade-off between the memory occupation and the speed of the solver, the more constraint we add the faster the solver gets but it also increases the memory occupation by new boolean variables.

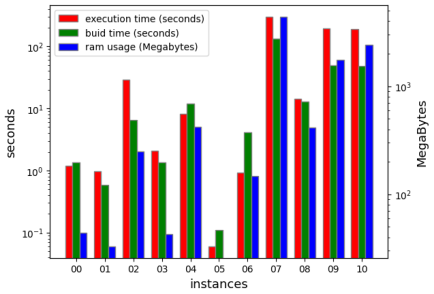
3.4.1 Experimental results

The outcomes derived from the employment of the SAT model have been documented in Table 2. Precisely, the tabulated data delineate the numeric outcomes

of the objective function pertaining to the initial eleven instances. For the subsequent cases, however, the notation "N/A" has been denoted, signifying an unattainable solution under the imposed temporal limitations.

ID	00	01	02	03	04	05	06	07	08	09	10	11-21
obj	**12**	**14**	**226**	**12**	**220**	206	322	536	**186**	**436**	**282**	N/A
time	2.64	1.39	47.39	1.77	4.18	0.02	0.93	301.3	48.38	192.23	14.29	300

Table 2: SAT results using z3 library.



The need for a custom encoding for each variable in SAT makes the solver very memory-heavy. We tried to minimize as much as possible the ram usage for our model. The data is presented in a logarithmic scale to improve readability.

Figure 3: execution, building and ram usage for SAT with Z3

4 Satisfiability Modulo Theories

The model is implemented in the Satisfiability Modulo Theories (SMT) framework using the $z3$ library. Exploiting the SAT-based nature of SMT, we enhance performance by making strategic use of boolean variables whenever feasible. To achieve this, boolean variables are converted to integers using the $z3.If(variable, 1, 0)$ function.

4.1 Decision Variables

Although based on the same core model as SAT, SMT implementation differs slightly due to the exclusion of the *courier_load* variable. This needed adapting constraint handling. Here are the key decision variables we employed:

- *table*: A 3-dimensional array of size $m \times (n+1) \times (n+1)$. Each element $table_{k,i,j}$ is a $z3.Bool$ variable that evaluates to true if courier k travels from position i to position j.
- *courier_distance*: A list of length m used to store the distances traveled by each courier. $courier\_distance \in [0, max\_path]$
- u: A 2-dimensional array of size $m \times (n+1)$ containing integer variables within the domain $[0, n]$.

4.2 Constraints

Our model incorporates a range of constraints to ensure feasible solutions that respect the problem's requirements. As the model parallels the SAT version, we'll focus on the differences:

- Each item is delivered precisely once. We express this constraint using the $z3.PbEq$ function to assert that exactly one out of $m \times (n+1)$ entries is true for both rows and columns:

$$\forall i \in \{0, \ldots, n\}, \sum_{k=0}^{m-1} \sum_{j=0}^{n} table_{k,j,i} = 1 \tag{8}$$

$$\forall i \in \{0, \ldots, n\}, \sum_{k=0}^{m-1} \sum_{j=0}^{n} table_{k,i,j} = 1 \tag{9}$$

- The cumulative size of items assigned to a courier must not exceed their maximum load capacity.

$$\forall k \in \{0, \ldots, m-1\}, \sum_{i=0}^{n} (\sum_{j=0}^{n-1} table_{k,i,j}) size_i \leq max\_load_k \tag{10}$$

- We utilize the following inequality to establish route ordering and prevent sub-tours [6] :

$$\forall k \in \{0, \ldots, m-1\}, \forall i \in \{0, \ldots, n-1\}, \forall j \in \{0, \ldots, n-1\},$$

$$u_{k,j} - u_{k,i} \leq 1 - (n+1) \times (1 - table_{k,i,j}) \tag{11}$$

To fully exploit the strengths of the theory solvers, all constraints expressed in boolean logic have been also rephrased using numeric variables.

4.2.1 Implied Constraints

While not strictly required by the problem, the following constraints are introduced to speed up the optimization process:

- Couriers must not stay in the same position.

$$\forall k \in \{0, \ldots, m-1\}, \forall i \in \{0, \ldots, n\}, table_{k,i,i} = 0 \tag{12}$$

- If a courier goes from i to j, it cannot go from j to i, except for the origin.

$$\forall k \in \{0, \ldots, m-1\}, \forall i \in \{0, \ldots, n-1\}, \forall j \in \{0, \ldots, n-1\}$$

$$\neg(table_{k,i,j} \wedge table_{k,j,i}) \tag{13}$$

- Each courier must visit at least a minimum number and at most a maximum number of positions

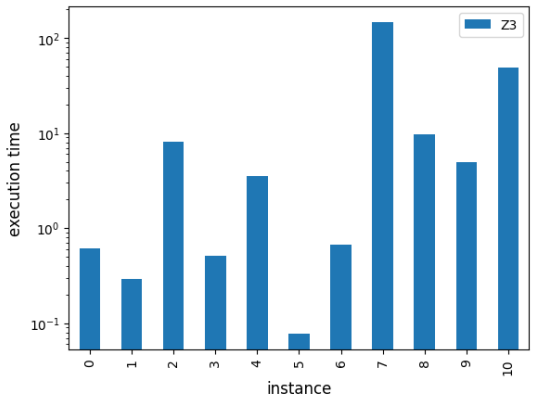
$$\forall k \in \{0, \dots, m-1\}, min\_packs \leq \sum_{i=0}^{n} \sum_{j=0}^{n-1} table_{k,i,j} \leq max\_packs \quad (14)$$

4.3 Validation

The outcomes derived from the employment of the SMT model have been documented in Table 3. Precisely, the tabulated data delineate the numeric outcomes of the objective function of the initial eleven instances. For the subsequent cases, however, the notation "N/A" has been denoted, signifying an impossible resolution under the imposed temporal limitations. In particular, the values found for the first eleven instances represent the value taken by the optimal solution.

ID	00	01	02	03	04	05	06	07	08	09	10	11-21
obj	**12**	**14**	**226**	**12**	**220**	**206**	**322**	**167**	**186**	**436**	**244**	N/A
time	0.55	0.28	8.25	0.59	1.70	0.66	146.2	9.81	9.81	5	48.5	300

Table 3: SMT results using z3 library.



SMT performs better when multiple constraint encodings are mixed (also repeating the same constraint in multiple ways helps), In such a way as to utilize the theory solvers as much as possible in order to increase speed. However, this approach makes the model harder to read. The data is presented in a logarithmic scale to improve readability.

Figure 4: Time diagram for SMT model in Z3

5 Mixed Integer Programming

In this section, we present the optimization model that utilizes Mixed Integer Programming (MIP) as the underlying optimization approach for solving the MCP problem.

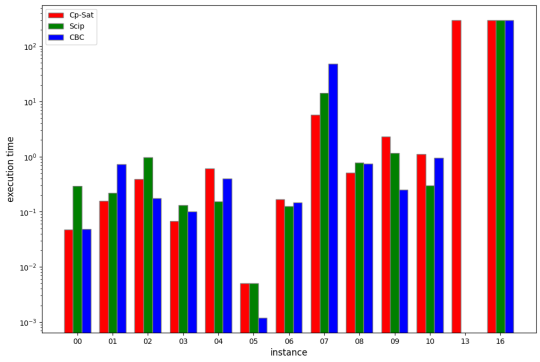
5.1 Decision Variables

The MIP implementation shares the same decision variables as the SMT model presented earlier. Please refer to the previous section for a detailed discussion of these variables.

5.2 Constraints

The MIP model also shares the same constraints as the previous models, although with some modifications. Specifically, for constructions, we represent these constraints using numeric values rather than boolean variables. We have worked to efficiently linearize these constraints to ensure a suitable formulation for the MIP solvers.

5.3 Validation



Of the various solvers, SAT is the best and most consistent one even though, for smaller instances, CBC may be faster 5. It's important to note that the data in the graph is presented on a logarithmic scale to enhance readability.

Figure 5: Time diagram for MIP solvers

At first different libraries for MIP have been tried (Pulp. Or-tools, Python-mip) to compare the different approaches using the same underlying solver (CBC). However, in the end, we decided to utilize the Or-tools MIP library to test different solvers (CBC, SAT, SCIP). Initially, our model incorporated a warm-start strategy utilizing the Clarke and Wright Savings Algorithm [9]. This algorithm was used to initialize the routes for the couriers based on an initial solution found by the heuristic. However, after evaluation, we determined that this strategy did not significantly improve the execution time of the model, and thus, we decided not to use it.

ID	SAT	CBC	SCIP
00	**12**	**12**	**12**
01	**14**	**14**	**14**
02	**226**	**226**	**226**
03	**12**	**12**	**12**
04	**220**	**220**	**220**
05	**206**	**206**	**206**
06	**322**	**322**	**322**
07	**167**	**167**	**167**

ID	SAT	CBC	SCIP
08	**186**	**186**	**186**
09	**436**	**436**	**436**
10	**244**	**244**	**244**
11-12	N/A	N/A	N/A
13	520	N/A	N/A
16	1102	460	496
17-21	N/A	N/A	N/A

Table 4: Comparison between the or-tools MIP model using different solvers

6 Conclusion

In conclusion, when comparing the different approaches for solving optimization problems, each method has its strengths and weaknesses, making the choice dependent on the specific problem characteristics. Constraint Programming is a highly flexible and efficient paradigm, often providing fast solutions. However, it heavily relies on global constraints. While this is not a drawback for well-studied problems like the one considered here, it might pose challenges for less-established problem domains. Mixed-Integer Programming offers a straightforward and fast approach, but it requires designing models with numerous variables, which can lead to substantial memory consumption. Its linearity is both an advantage and a limitation. Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) solvers are not primarily designed for optimization tasks. SAT, in particular, demands encoding the problem, which adds complexity to the model design process. SMT is more user-friendly than SAT but still may not be the best choice for optimization due to memory constraints and other limitations. Ultimately, the choice of the most suitable approach depends on the specific problem's nature, complexity, and the availability of domain-specific knowledge. It's essential to weigh the pros and cons of each method carefully and consider factors such as efficiency, memory requirements, and the problem's unique characteristics when selecting the appropriate optimization solver.

References

[1] Hoffman, Karla & Padberg, Manfred. (2001). Traveling salesman problem. 10.1007/1-4020-0611-X_1068.

[2] G. Gutin, & The Traveling Salesman Problem and Its Variations

[3] Benchimol, P., Hoeve, WJ.v., Régin, JC. et al. Improved filtering for weighted circuit constraints. Constraints 17, 205–233 (2012). https://doi.org/10.1007/s10601-012-9119-x

[4] Bierlee, H., Gange, G., Tack, G., Dekker, J.J., Stuckey, P.J. (2022). Coupling Different Integer Encodings for SAT. In: Schaus, P. (eds) Integration of Constraint Programming, Artificial Intelligence, and Operations Research. CPAIOR 2022. Lecture Notes in Computer Science, vol 13292. Springer, Cham. https://doi.org/10.1007

[5] Searching and Sorting algorithms https://codeburst.io/algorithms-i-searching-and-sorting-algorithms-56497dbaef20

[6] Martin Desrochers, Gilbert Laporte, Improvements and extensions to the Miller-Tucker-Zemlin subtour elimination constraints, Operations Research Letters, Volume 10, Issue 1, 1991, Pages 27-36, ISSN 0167-6377, https://doi.org/10.1016/0167-6377(91)90083-2.

[7] A new constraint programming model and a linear programming-based adaptive large neighborhood search for the vehicle routing problem with synchronization constraints, Computers & Operations Research, volume 124, year 2020, ISSN 0305-0548, DOI https://doi.org/10.1016/j.cor.2020.105085, URL https://www.sciencedirect.com/science/article/pii/S0305054820302021

[8] Official minizinc docker image https://hub.docker.com/r/minizinc/minizinc#!

[9] G. Clarke, J. W. Wright, (1964) Scheduling of Vehicles from a Central Depot to a Number of Delivery Points. Operations Research 12(4):568-581. https://doi.org/10.1287/opre.12.4.568