The Multiple Couriers Planning Problem

Calin Diaconu - calin.diaconu@studio.unibo.it
Maxence Murat - maxence.murat@studio.unibo.it

May, 2024

1 Introduction

The following report describes our solution to the multiple couriers planning problem, or as it is sometimes called in literature, the vehicle routing problem. This document is structured in 6 parts, with the current introduction aiming to describe the common aspects of the 4 implemented solutions.

All solutions have the same input parameters, as described in the requirements document: the number of couriers (m), the number of items (n), the number of locations (one more compared to the number of items, since an origin point is also considered), an array of integers representing the maximum load of each courier (l, which represents the total maximum size of all the items the courier can carry), an array of integers representing the size of each item (s), and a matrix storing the distances between locations (D).

In this report, the constants for each specification in the data, like the number of couriers, will be referred to by the notation given in the project requirements document and as described above (e.g., the number of couriers is written as m).

Regarding **the common variables, constraints, and bounds**, a general model has been compiled, which was inspired by solutions to the vehicle routing problem found in literature or online ([1], [2], [3], [4], [5], [6]). On the topic of variables, one- or multi-dimensional arrays will represent the assignment of items/locations to couriers and the existence of precedence between any two locations. A variable is required for subtour elimination, implemented through the Miller-Tucker-Zemlin (MTZ) formulation [7], which makes sure that at each no cycles exist in the path of a courier. The final variable worth mentioning is the objective one, namely the length of the longest route of any of the couriers. As for the constraints, the following common ones are present:

1. Exactly one visit per location by any and all couriers:

$$\forall i \in [1, n], (\sum_{j=1}^{m} \text{item\_i\_assigned\_to\_courier\_j}) == 1 \quad (1.1)$$

$$\forall i \in [1, n], (\sum_{j=1}^{n} \text{item\_i\_predecessor\_of\_item\_j}) \leq 1 \quad (1.2)$$

2. Each courier carries at most the maximum load defined for itself (assume that the truth values correspond to 0 and 1 integer values):

$$\forall j \in [1, m], \sum_{i=1}^{n} (\text{item\_i\_assigned\_to\_courier\_j} * s[i]) \leq l[j]$$

3. The MTZ constraint:

$$\forall i, j \in [1, n],\ \text{item\_i\_predecessor\_of\_item\_j} == 1 \implies$$

$$\text{steps\_from\_origin\_to\_reach\_delivery\_point\_of\_item\_j} ==$$

$$(\text{steps\_from\_origin\_to\_reach\_delivery\_point\_of\_item\_i} + 1)$$

4. Each courier must begin and end route in origin:

$$\forall i \in [1, m],\ \text{initial\_location\_of\_courier\_i} == \text{final\_location\_of\_courier\_i} == (n+1)$$

5. At every step, any courier must move to a new location, which is equivalent to no 2 items have the same precedent:

$$\forall i, j, k \in [1, n],\ \text{item\_i\_predecessor\_of\_item\_j} == 1 \implies \text{item\_i\_predecessor\_of\_item\_k} == 0$$

6. An item that precedes a second one as delivery order must be carried by the same courier as the second item:

$$\forall i, j \in [1, n],\ \text{item\_i\_predecessor\_of\_item\_j} == 1 \implies \exists k \in [1, m] \text{ s.t.}$$

$$\text{item\_i\_assigned\_to\_courier\_k} == \text{item\_j\_assigned\_to\_courier\_k} == 1$$

For the current project, one could talk about two different **pre-processing steps**. The first one is specific to the CP solution, and refers to the transformation of the input data of each instance in the MiniZinc specific format, a dzn file. The second one is specific to the MIP solution, and refers to the nearest neighbour heuristic, used for warm start searches. This heuristic cannot be applied to the other 2 models due to different limitations: while CP has a warm start annotation for searches, it is not actually implemented for the solver that were used here; as for SAT, this type of model doesn't support this kind of a search.

Since **symmetry breaking** can dramatically reduce the required time to find a solution, in the initial phase of the project, an analysis on this has been made, using a short, custom Pythons script. Given the definition of the problem, three cases of symmetry can be observed: symmetry on couriers load capacity, when two of them have an equal capacity, on items, when two of them have the same size, and reflexivity on the distances between locations, where the distance between two locations is equal in both directions for any pair of locations. The problem instances in which each of these 3 symmetries appears are shown in Table 1.

While the courier size symmetry actually happens and can be fixed, the item size one can not actually be defined as a symmetry, because the distance from the delivery points of the 2, to any other location, may be different.

As for location distance reflexivity, it can not be simply solved by avoiding the inversion of 2 items, since again, the distance from either of the 2 to any other item may be different. However, this causes a symmetry in the case of a path on multiple items that present this reflexivity, in the way that, if all the locations are passed through in regular order, or in reversed order, the total distance will be the same. Thus, this particular case, where out of the 2 reversed orderings only one is verified, can be considered in the constraints.

Regarding the **difficulties** we encountered, all these three solutions were implemented using programming paradigms that were new to us. They may resemble the structure of logic programming in the sense that, at least in the case of CP, it is a declarative language, but is considerably different from that one as well. So it implied the need of a different way of thinking, and a different overall approach in the design of the solutions. Other difficulties we encountered were a relative lack of online resources (compared to more popular paradigms, like sequential programming), and a rather high time required for thoroughly testing different configurations (worst case scenario meant 5 minutes per test, with 21 total tests, so 105 minutes for all tests). Also, in terms of oragnization, we started as a team of 4 students, but 2 of us had to drop out, due to different personal reasons.

After the changes in the team line up, we decided to implement solutions for all 4 types of models. Maxence worked on SMT and MIP, while Calin developed models for CP and SAT. It took us a total of about 3-4 months of non-continuous work to finish the 4 implementations.

On the more **technical side**, the project was implemented in Python, thus having a unified access point for all solving methods. This language has been chosen first of all because of its flexibility, ease of use, and of our familiarity with it, and secondly because some of the needed solvers, such as Z3, are available as Python libraries. A main pipeline was created, which can be accessed through a shell script, with the instance to be tested and the solver to be applied being changeable through parameters. More instructions on how to do it are available in the README file.

Instance ID	Location Distance	Courier Load	Item Size
1	F	F	T
2	F	T	T
3	F	F	T
4	F	T	T
5	F	F	F
6	F	T	T
7	F	T	T
8	F	T	T
9	F	T	T
10	F	T	T
11	T	T	T
12	T	T	T
13	T	T	T
14	T	T	T
15	T	T	T
16	T	T	T
17	T	T	T
18	T	T	T
19	T	T	T
20	T	T	T
21	T	T	T

Table 1: The symmetries table, where T denotes that the type of symmetry happens for the specified instance, and F otherwise.

2 The Constraint Programming (CP) Model

According to [8], CP is a paradigm for solving combinatorial problems by declaratively stating constraints on the feasible solution for a set of decision variables. In this section, the details of the CP solution for the problem at hand will be described.

2.1 Decision Variables

The decision variables used for the CP model, with their initial domains and meaning, are:

- *courier_assignment* - an array of n elements, that has the domain assigned to COURIERS, which is the set [1..m]; courier_assignment[it] = co iff item *it* is delivered by courier *co*;
- *item_assignment* - an m*n matrix of booleans; item_assignment[co, it] = true iff item *it* is delivered by courier *co*;
- *pre* - an array of n elements, that has the domain assigned to LOCATIONS (so it includes all the ids for the items, where *i* corresponds to the delivery location of the item with id *i*, plus an extra one for the origin), which is the set [1..n+1]; pre[it1] = it0 if item *it0* is delivered immediately before item *it1*, by the same courier, or pre[it] = (n + 1) if item *it* is the first one delivered by its assigned courier;
- *pre_table* - an n*n matrix of booleans; pre_table[it1, it2] = true if (pre[it1] = it2) or (it1 == it2 and this is the first item delivered by its assigned courier), and false otherwise;
- *steps_from_origin* - an array of n elements, that has the domain assigned to ITEMS, which is the set [1..n]; steps_from_origin[it] = x, where *x - 1* is the number of all items delivered by the assigned courier before item *it*, so the domain is set to be as tight as possible, but to also take into consideration the worst case, where all items are delivered by a single courier;
- *covered_distances* - an array of m elements, that has the domain assigned to [0..x], where *x* is total distance covered by a single courier in the worst case, when all items are delivered by a

single courier (a sum of the worst distance from each item to another, and the longest distance needed to get from an item to the origin); covered_distances[co] = x iff x is the distance needed by courier *co* to deliver all the items assigned to them, in the order given in the *pre* array.

2.2 Objective Functions

The objective variable is *max_dist*. Its domain is the same as the domain of each item in *covered_distances*, as described in the previous subsection. Its value will always be the maximum element of the *covered_distances* array. The search algorithm is set to minimize this variable, since the aim is to obtain the smallest possible maximum distance covered by any of the couriers.

Another variable that is relevant for the search is *pre_dist*. It is designed to help the search algorithm in minimizing the longest distance by starting with an order of items that yields the smallest values possible for the distances between the items. Thus, the domain is set to be as tight as possible, namely to the set of all the possible distances between any 2 locations. The value of pre_dist[it1] is D[it0, it1] iff pre[it1] = it0.

2.3 Constraints

Below, the constraints imposed in the solution are described.

Main Constraints

Since each courier can be seen as a bin containing all the items delivered by them, we can enforce the maximum capacity of each courier through *bin_packing_capa*, a global constraint provided by the MiniZinc language. The arguments taken by it are, in this order, an array containing the maximum capacity of each bin, an array marking the assignment of each item to a bin, and an array containing the size of each item. This is equivalent to constraint 2 from the introduction.

The distances between 2 successively delivered items are taken from the D matrix and marked in the *pre_dist* array using a *forall* iteration and an equality check.

Because two representations are used for how the items are assigned to the couriers, a channeling constraint is used between the *item_assignment* matrix and the *courier_assignment* array.

A similar situation happens between the *pre_table* matrix and the *pre* array. However, the channeling here happens in multiple steps, because of the different conditions for when an item is the last one to be delivered by a courier. The true cells are marked through 2 equality constraints, and the false ones are marked through another one, by ensuring that on a column (where column i corresponds to the predecessor of item i), the sum of elements is 1. Thus, a single item will be assigned in the matrix as a predecessor to another one (as in constraint 5 in the introduction).

This last condition is enforced in the *pre* array by an *alldifferent_except* constraint. This checks that any location is the predecessor of exactly one item, with the exception of the origin (since for the first item of each courier the predecessor will be marked with *n + 1*, which is the id of the origin location). Another constraint for the number of items that have as predecessor the origin is enforced through a *count* statement applied on the *pre* array. This is equivalent to constraints 1 and 4 from the introduction.

Another obvious condition is that precedence can only happen between items delivered by the same courier. This constraint is performed through an equality between the courier assigned to item *it*, and to the one assigned to its previous item (*pre[it]*). The items that are to be the first ones delivered by a courier are skipped. This is an adaptation of constraint 6 from the introduction.

Another important condition is that there are no cycles when a courier delivers their items. More precisely, there can not be any number *a* of items where *pre[it_a] = it_(a-1), pre[it_(a-1)] = it_(a-2), ..., pre[it_2] = it_1, pre[it_1] = it_a*. The two constraints used for this are inspired by how the tree predicate is implemented in the MiniZinc language. The delivery schedule can be represented as a graph (as represented in the *pre* array, without marking the step that each courier has to do from the last item to the origin), that in the end should be a tree, with the root in the origin, and each branch being the path taken by a courier (thus all nodes will have a single unique child, and a single unique parent, with the exception of the origin, which has 0 parents and a number of children equal to the number of couriers used in the solution). There is an equality verified in each of the 2 constraints, both operating on the *steps_from_origin* variable. They ensure that all items that are the first ones

delivered by each of the couriers have the value assigned to 1 in this array, and $x + 1$ otherwise, where x is the number of steps required for the previous item. This is equivalent to the third constraint from the introduction.

Lastly, two constraints are imposed on the objective value and the array from which it is extracted, in order to compute the distances covered by each courier, and the maximum value out of these.

Implied Constraints

The first implied constraint checks that each item is carried by a single courier, by checking that there is only one cell for each item having a true value in the item_assignment matrix. This is already verified, since the courier_assignment array has a single integer value, representing the id of a single courier, for each cell (corresponding to an item).

The second one checks that in the *pre* array no item precedes itself. This is already taken care of through the cycle breaking constraint and through the channeling between the predecessor table and predecessor array.

The third such constraint does a global *alldifferent* on the *steps_from_origin* items that belong to the same courier, such that no items delivered by the same courier require the same number of steps. This is previously solved by the definition of this array itself, since the value in each cell is an increment of another one from the array.

The last implied constraint makes sure that each item is the predecessor of either no other item in the *pre* array, if it is the last one delivered by a courier, or by a single other item otherwise. This is implied by all the conditions previously applied on the *precedence_table*.

These 4 constraints should cut down the search time by removing some search cases, but as it is mentioned in the Validation subsection, they do not help in all the test cases.

Symmetry Breaking Constraints

As described in the Introduction, there are only 2 symmetries that can be broken.

For the one related to the maximum load of a courier, a *lex_less* constraint is applied on the *item_assignment* matrix, such that no identical assignment of elements is given to 2 identical couriers.

As for the symmetry related to the distances between items, a *lex_lesseq* constraint is applied on the *pre_table* matrix, such that the reversed order of delivery is avoided. For example, if for any 2 items, it1 and it2, *D[it1, it2] = D[it2, it1]*, and we consider 3 items, a, b, and c, only one of the orderings a - b - c and c - b - a will be considered, since the distances covered in the 2 situations are identical.

2.4 Validation

The results of the experiments are reported in Figure 1 for the execution times, and in Tables 2, 3, and 4, for the objective values, where the notations have the same meaning as the problem requirements. The only difference is N/S, which means no symmetry, and is used in the cases where a model was not tested on a certain instance, since that symmetry is not present in that instance, as detailed in the introduction. The omitted cells contained N/A values and were skipped for brevity. Each model and search strategy mix is noted with M and a number, having the meanings below:

- M1, M2, and M3 - represent the baselines using the Gecode, Chuffed, and OR Tools CP-SAT; they are equivalent to *baseline_gecode*, *baseline_chuffed*, and *baseline_lcg* respectively in the result files;
- M4, M5, and M6 - are similar to the previous 3 setups, with the differences being that the redundant constraints are not ignored; they are equivalent to *redundant_gecode*, *redundant_chuffed*, and *redundant_lcg* respectively in the result files;
- M7 and M8 - are similar to the first 2 setups, with the difference being that the symmetry breaking constraints for courier capacity are included this time; they are equivalent to *symmetries1_gecode* and *symmetries1_chuffed* respectively in the result files;
- M9 and M10 - are similar to the first 2 setups, with the difference being that the default search is replaced with int_search, applied on the following variable, in this order: pre_dist, smallest,

and indomain_split; they are equivalent to *int_search_gecode* and *int_search_chuffed* in the result files;

- M11 and M12 - are similar to the first 2 setups, with the difference being that the default search is augmented with luby restart, with the limit set to 250; they are equivalent to *restart_gecode* and *restart_chuffed* in the result files;
- M13 and M14 - are similar to the first 2 setups, with the difference being that the default search is augmented with relax_and_reconstruct applied on pre_dist, with the limit set to 50; they are equivalent to *lns_gecode* and *lns_chuffed* respectively in the result files;
- M15 and M16 - have all the modifiers described so far applied on the first 2 setups; they are equivalent to all_without_s2_gecode and all_without_s2_chuffed in the result files;
- M17 - is similar to M15, with symmetry breaking included also for the distance symmetry; it is equivalent to all_with_s2_gecode in the result files;
- M18 - is similar to M17, with the difference being that the parameter of the luby restart is lowered to 50; it is equivalent to smaller_restart_gecode in the result files;
- M19 - is similar to M18, with the search strategy changed from int to random; it is equivalent to smaller_restart_random_gecode in the result files;
- final_config - is the configuration that is included in the repository (Gecode, with both symmetries broken, with int_search having the same setup as in M9, linear restart with the parameter set to 50, and relax_and_reconstruct on pre_dist, with the parameter set to 80).

There is a rather high number of experiments as a result of the attempt to evaluate what influence each individual change has on the results. The final_config was considered as such since instance 13 was taken as reference, among the more difficult cases, and it obtained the best score with this combination of model and search strategy. However, it's worth to note that overall, M9 obtained solutions for more instances (16 and 19, beside 13), with a comparably low distance for instance 13.

On the topic of the hardware used, the experiments were performed on a laptop with a 3.3 GHz, 6-core, 12-threaded CPU, with 15.4 GB of available RAM in total and an SSD for storage. Everything was run through a Docker container, and it is worth mentioning that the times obtained in the native system were lower than when run through the container, as expected since the Docker system can be computationally expensive. Regarding the solvers, for Gecode the 6.3.0 version was used, 0.13.1 for Chuffed, and 9.8.3296 for OR Tools CP-SAT. The random seed was set to 42.

Analyzing the results, on the topic of solvers, one could notice that OR Tools was dropped quickly. This was done because it was obtaining similar results to Chuffed, both in terms of objective value and time, and thus time was saved when running experiments. As expected because of how are designed, Chuffed gives very good results for small instances, up to instance 10, reaching optimality even with default search on almost all of them. However, Gecode performs better on larger instances, when Chuffed doesn't even manage to obtain a solution. Almost all alterations obtain better objective values on almost all instances, with a combination and re-tweak of all of them getting the best objectives.

On almost all large instances, from 11 up, the setups don't manage to reach solutions, with the exception of 13, 16, and 19. 13 is the smallest in this subset of instances, with just 3 couriers, the others having 20 each. 16 and 19 have the smallest number of items, with 47 and 71 each. When following the execution, one can observe that regarding the speed to reach an initial solution, the random search is the fastest, as expected, since through many random trials, a good configuration can be found. Luby restart obtains very good results on all three of the big functional instances, while a combination of all the improvements and tweaks to them obtains the smallest result on instance 13.

3 The SAT Model

According to [9], boolean/propositional satisfiability problems (SATs) refer to the process of checking whether there exists an interpretation that satisfies a given boolean formula. Below, the implementation and experiment results of an SAT setup for solving the multiple couriers planning problem are described.

3.1 Decision Variables

There are less decision variables employed for this model, since no integers could be used. Thus, only the boolean tables necessary to describe a solution and the objective value expressed with booleans in base 2 were kept. The integer variables and the associated constraints could be expressed through base 2 operations, but the constraint declaration time is already high, so they have been skipped.

The lengths of the binary numbers are computed dynamically for each instance and for each number they represent, depending on the overall maximum base 10 value in the given instance of each type of number (weight, number of steps, or distance).

The variables have the following meanings:

- *item_assignment* - an m*n matrix of booleans, where item_assignment[co][it] == true iff item *it* is deliverd by courier *co*;
- *pre_table* - an n*n matrix matrix of booleans, where pre_table[it1, it2] == true iff item *it1* is immediately preceded by item *it2* (which can also be interpreted as the delivery order being it2 →it1, with no other item being delivered between them);
- *steps_from_origin* - an n*steps_len matrix, where row *i* is the base 2 representation of the number of steps needed by a courier to get from the delivery location of the first item delivered by them to the delivery location of item i, given the route in the current solution;

3.2 Objective Functions

While SAT can not be used to solve a regular optimization problem, since it doesn't deal directly with integers, here the *max_dist* variable was used to impose a limit on the objective value as it was described in the requirements. It is a binary representation of the base 2 maximum distance, through an array of booleans of a dynamically computed length, as described previously. Aiming to minimize this value, after every solution is found, a new constraint is added to the model that imposes max_dist to be smaller than the previously found solution, with no limit imposed to it on the first iteration.

3.3 Constraints

Due to the nature of the implementation, the constraint declaration is slower than for the other models. While they should not be taken into consideration towards the time limit of 5 minutes, a 10 minute ending point was established for this part. Out of the large instances, from 11 onward, just the smallest of them (13) managed to finish constraint declaration and to get to the solving part, with no solution being yielded in the 5 minute execution time frame. Below, the constraints imposed in the SAT solution are described.

Main Constraints

In an attempt to reduce the constraint declaration time, the courier capacity limit enforcement and the computation of the objective value are performed in the same loop. Auxiliary variables are used to store the list of carried weights and distances covered by each courier in base 2 representation, from which the sum of total weights and the maximum such distance are extracted. The sum of the weights is set to be less or equal to the maximum load of each courier, given through the instance parameters, equivalent to constraint 2 from the introduction.

For the same purpose, the remaining constraints are imposed in the same loop or, where possible, in the same constraint. Constraint 5 from the introduction is achieved both through forbidding an item to precede itself, and 2 different items having the same precedent if the precedent is not the origin (through the check on pre_table row of it1 to be different from 0), as follows:

$$\forall it \in [1, n], !pre\_table[it][it] == true$$

$$and$$

$$\forall it1 \in [1, n], it2 \in (it1, n], (\exists iin[1, n]s.t.pre\_table[it1][i] == 1) \implies$$

$$(\exists j \in [1, n]s.t.pre\_table[it1][j]! = pre\_table[it2][j])$$

MTZ is implemented in a similar manner as described in the introduction, using the steps_from_origin variable. Since for the second item the iteration starts from the id of the first one, such that a number of redundant loops is removed, both forward and backward checks are performed. Constraint 1 from the introduction is obtained by making sure exactly one courier is assigned to each item in the item_assignment table. Constraint 6 is also achieved through a forward and a backward check , by comparing the pairs of courier assignment rows for all 2 items that precede each other. Constraint 4 is enforced through an implication with 2 sub-conditions.

Implied Constraints

As explained before, the problem is represented through less variables than other cases, the integer ones being avoided as much as possible. Since the concepts of the problem, such as precedence, are only expressed in a single manner now, no implied constraints are involved any more.

Symmetry Breaking Constraints

The **maximum load symmetry** is broken for each pair of couriers by ensuring that the item assignment is in the same lexicographic order as the courier ids. This is easily achieve through a custom less-or-equal implementation, applied on binary arrays that can be interpreted also as base 2 numbers. The **distances symmetry** is broken in a similar fashion, but with the comparison applied on the pre_table variable.

3.4 Validation

The experiment results are reported in Table 5. Only the first 10 instance appear, since for the following ones no solution was found, with the distance symmetry breaking being consequently omitted. The values in the table have the same meaning as talked about in the requirements, with *NO_SYMM* meaning that the load symmetry doesn't apply to that instance, so it was not tested.

The same hardware configuration was used as for CP, with the same mention about execution time for between Docker and no-Docker executions. The version used for the Z3 library was 4.13.0.0.

Regarding the results, one can notice that the performance of all 3 setups for the SAT model is inferior to others. Optimality is checked only for the simplest 2 instances, 1 and 3, even though is achieved for all instances, with or without symmetry breaking, as cross-checked with the results of the other models. The load symmetry breaking only helps on instance 7, which doesn't achieve an optimal value in both cases, having an objective value significantly higher compared to other methods. Time is not mentioned in a separate figure since, in the few cases of achieving optimality, the solution only needed one or two seconds. For the larger instances, as mentioned before, after 10 minutes, constraint declaration concludes only for instance 13, but it doesn't reach a solution in the required time. Longer solving times, up to one hour, have also been tried for instance 13, but the model still did not return a solution.

4 The SMP Model

4.1 Decision Variables

The model introduces several decision variables:

Couriers Transfer Variables

$$x_{i,j,k} \in \{0,1\}, \quad \forall i \in m, \forall j \in n, \forall k \in n$$

Binary variable is assigned value = 1 if courier i travels directly from location j to location k, including the origin.

Courier Visit Variables

$$y_{i,j} \in \{0,1\}, \quad \forall i \in m, \forall j \in n$$

Binary variable indicating whether courier i has visited location j. Initialized with appropriate domains.

Integer MTZ Variables

$$u_{i,j} \in \mathbb{Z}, \quad \forall i \in m, \forall j \in n$$

Integer variable used in the Miller-Tucker-Zemlin (MTZ) formulation for sub-tour elimination.

Objective Function Variable

$$\text{max\_distance} \quad \text{(Integer)}$$

The variable to be minimized in the objective function, representing the maximum distance any courier has to travel.

4.2 Objective Function

The objective function aims to minimize the maximum distance traveled by any courier. This is expressed as:

$$\min \text{max\_distance}$$

Subject to the constraint:

$$\sum_{j=0}^{n} \sum_{k=0}^{n} x_{i,j,k} \cdot D_{j,k} \leq \text{max\_distance}$$

4.3 Constraints

Assignment Constraints

Each location must be assigned to exactly one courier:

$$\sum_{i=1}^{m} y_{i,j} = 1 \quad \forall j \in n$$

Linking Constraints

Links the transfer variable to the visit variable. Limits each courier to exactly one single visit variable per location so that if any courier i goes to or leaves location j, it takes on this location's load:

$$\sum_{k \neq j} x_{i,j,k} = y_{i,j} \quad \forall i \in m \ , \forall j \in n$$

$$\sum_{j \neq k} x_{i,j,k} = y_{i,k} \quad \forall i \in m \ , \forall k \in n$$

Load Capacity Constraint

Each courier's load capacity must not exceed their specified limit:

$$\sum_{j=1}^{n} s_j \cdot y_{i,j} \leq l_i \quad \forall i \in m$$

MTZ Constraint

Miller-Tucker-Zemlin constraint ensures a coherent tour for each courier that does not cycle through subtours:

$$u_{i,j} - u_{i,k} + n \cdot x_{i,j,k} \leq n - 1 \quad \forall i \in m \ , \forall j \in n \ , \forall k \in n \ , \text{ where } j \neq n \text{ and } j \neq k$$

MTZ explanation: Simply explained, the MTZ constraint assigns a value u to a courier upon arrival to a location. The courier is only allowed to transfer to a location that has a greater u value than the u value of the current location. Each successive u value is incremented so that a courier can not return to a previous location, since u values are by definition lower than the most recent one.

Courier Origin Constraints

Each courier must leave the origin and return to it exactly once:

$$\sum_{k=0}^{n} x_{i,n,k} = 1 \quad \forall i \in m$$

$$\sum_{j=0}^{n} x_{i,j,n} = 1 \quad \forall i \in m$$

4.4 Validation

Experimental Design

The experimental design follows the same procedure as the MIP program, with the exception that it does not warm start the program with a nearest-neighbor heuristic.

Reproducibility Information For reproducibility's sake, here is the hardware and software employed in the Google Colab environment:

Hardware:

- 2vCPU Intel(R) Xeon(R) CPU @ 2.20GHz
- 13GB RAM
- 100GB Free Space

Software:

- Z3 Version 4.13.0.0

Time Limit:

- 180 seconds/3 minutes for constraints and 300 seconds/5 minutes for solver

5 The MIP Model

5.1 Decision Variables

The model introduces several decision variables:

Couriers Transfer Variables

$$x_{i,j,k} \in \{0, 1\}, \quad \forall i \in m, \forall j \in n, \forall k \in n$$

Binary variable is assigned value = 1 if courier i travels directly from location j to location k, including the origin.

Courier Visit Variables

$$y_{i,j} \in \{0,1\}, \quad \forall i \in m, \forall j \in n$$

Binary variable indicating whether courier i has visited location j. Initialized with appropriate domains.

Integer MTZ Variables

$$u_{i,j} \in \mathbb{Z}, \quad \forall i \in m, \forall j \in n$$

Integer variable used in the Miller-Tucker-Zemlin (MTZ) formulation for sub-tour elimination.

Objective Function Variable

$$\text{max\_distance} \quad \text{(Integer)}$$

The variable to be minimized in the objective function, representing the maximum distance any courier has to travel.

5.2 Objective function

The objective function aims to minimize the maximum distance traveled by any courier. This is expressed as:

$$\min \text{max\_distance}$$

Subject to the constraint:

$$\sum_{j=0}^{n} \sum_{k=0}^{n} x_{i,j,k} \cdot D_{j,k} \leq \text{max\_distance}$$

5.3 Constraints

Assignment Constraints

Each location must be assigned to exactly one courier:

$$\sum_{i=1}^{m} y_{i,j} = 1 \quad \forall j \in n$$

Linking Constraints

Links the transfer variable to the visit variable. Limits each courier to exactly one single visit variable per location so that if any courier i goes to or leaves location j, it takes on this location's load:

$$\sum_{k \neq j} x_{i,j,k} = y_{i,j} \quad \forall i \in m \ , \forall j \in n$$

$$\sum_{j \neq k} x_{i,j,k} = y_{i,k} \quad \forall i \in m \ , \forall k \in n$$

Load Capacity Constraint

Each courier's load capacity must not exceed their specified limit:

$$\sum_{j=1}^{n} s_j \cdot y_{i,j} \leq l_i \quad \forall i \in m$$

MTZ Constraint

Miller-Tucker-Zemlin constraint ensures a coherent tour for each courier that does not cycle through subtours:

$$u_{i,j} - u_{i,k} + n \cdot x_{i,j,k} \leq n - 1 \quad \forall i \in m \ , \forall j \in n \ , \forall k \in n \ , \text{ where } j \neq n \text{ and } j \neq k$$

MTZ explanation: Simply explained, the MTZ constraint assigns a value u to a courier upon arrival to a location. The courier is only allowed to transfer to a location that has a greater u value than the u value of the current location. Each successive u value is incremented so that a courier can not return to a previous location, since u values are by definition lower than the most recent one.

Courier Origin Constraints

Each courier must leave the origin and return to it exactly once:

$$\sum_{k=0}^{n} x_{i,n,k} = 1 \quad \forall i \in m$$

$$\sum_{j=0}^{n} x_{i,j,n} = 1 \quad \forall i \in m$$

Implied Constraint: Flow Constraints

Each courier, when it visits a location, must then move to visit another location

$$\sum_{k=0}^{n} x_{i,j,k} - \sum_{k=0}^{n} x_{i,k,j} = 0 \quad \forall i \in m \ , \forall j \in n$$

5.4 Validation

Experimental Design

The same MIP model was tested according to two varying search strategies; one with a nearest neighbor heuristic to warm start the search and one without. The nearest neighbor heuristic was meant to help tackle larger data instances because the program struggled to find initial solutions in such instances.

Reproducibility Information For reproducibility's sake, here is the hardware and software employed in the Google Colab environment:

Hardware:

- 2vCPU Intel(R) Xeon(R) CPU @ 2.20GHz
- 13GB RAM
- 100GB Free Space

Software:

- Gurobi 11.0.0

Time Limit:

- 300 seconds/5 minutes

5.4.1 Experimental Results

In all 21 data instances, the Gurobi Baseline model would always find an optimal solution faster than the Gurobi Heuristic-based model (at least when a solution was found). There could be 2 main reasons causing this:

- The initial heuristic solution may not necessarily be close to the optimal solution, hence could be leading the optimizing algorithm in the wrong direction. It is indeed possible that a non-optimal initial solution makes it harder to converge quickly to the final optimal solution.
- It is possible that the initial calculation of the nearest neighbor heuristic takes time to process. Processing the initial solution might constitute an overhead that is not significantly helpful, especially if the problem is already quickly solvable.

While the nearest neighbor heuristic failed to speed up the optimization process, especially in the first 10 instances of data, it works wonderfully in other cases wherein the Baseline model is unable to find even a non-optimal solution. Indeed, in instances 11 to 21, it is observed that without an initial solution to warm start the program, the Baseline model is unable to come up with any type of solution.

In the cases where the model was unable to find any working solution without a heuristic-based solution, it rarely could improve the initial solution by a significant amount. Essentially, this signifies that the heuristic may be performing almost too well relative to the MIP model.

6 Conclusions

Overlooking the entire project, one can observe that each solution comes with its particularities, its advantages and disadvantages. CP had the most options to configure, especially in terms of search strategies. With some tweaks, good results have been obtained for a number of the more difficult instances. SAT, while not designed for this, also manages to achieve optimal results on almost all easy instances of this optimisation task. SMT also gets to optimal results for simple tasks, and manages to get some solutions also for 2 of the more complex ones. MIP ends up getting the best results, thanks to the heuristic and the professional software used: solutions are found for almost all instances in the given time, with the exception of the most difficult instance, and the distances obtained are either optimal for 9 out of the 10 simpler tasks, or very good if compared to the other solutions for the more difficult instances.

References

[1] Wikipedia contributors. Vehicle routing problem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Vehicle_routing_problem&oldid=1225037791, 2024. [Online; accessed 29-May-2024].

[2] George B Dantzig and John H Ramser. The truck dispatching problem. *Management science*, 6(1):80–91, 1959.

[3] Avijit Basak and Subhas Acharya. Onsite job scheduling by adaptive genetic algorithm. *arXiv preprint arXiv:2306.02296*, 2023.

[4] Nafiz Mahmud and Md Mokammel Haque. Solving multiple depot vehicle routing problem (mdvrp) using genetic algorithm. In *2019 International Conference on Electrical, Computer and Communication Engineering (ECCE)*, pages 1–6. IEEE, 2019.

[5] J Christopher Beck, Patrick Prosser, and Evgeny Selensky. Vehicle routing and job shop scheduling: What's the difference? In *ICAPS*, pages 267–276, 2003.

[6] W David Fröhlingsdorf. Using constraint programming to solve the vehicle routing problem with time windows. 2018.

[7] Luis Gouveia. Using the miller-tucker-zemlin constraints to formulate a minimal spanning tree problem with hop constraints. *Computers & Operations Research*, 22(9):959–970, 1995.

[8] F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming*. ISSN. Elsevier Science, 2006.

[9] Wikipedia contributors. Boolean satisfiability problem — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Boolean_satisfiability_problem&oldid=1219369085, 2024. [Online; accessed 30-May-2024].

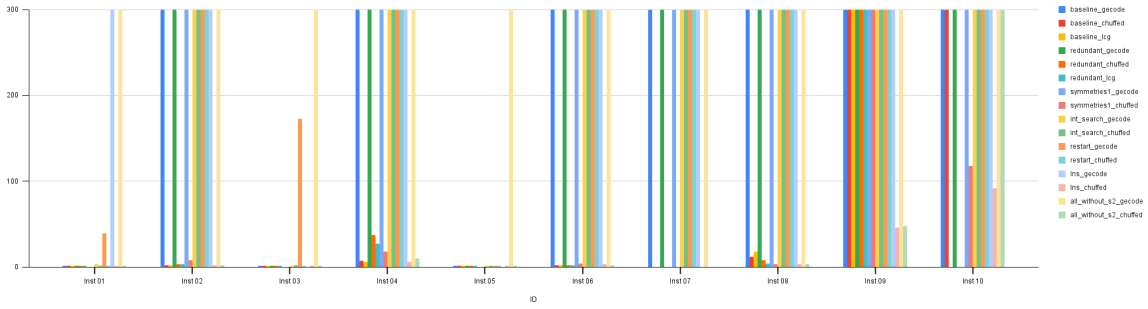

Figure 1: CP Running Times

ID	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12
01	**14**	**14**	**14**	**14**	**14**	**14**	N/S	N/S	**14**	**14**	**14**	**14**
02	311	**226**	**226**	277	**226**	**226**	277	**226**	245	240	311	270
03	**12**	**12**	**12**	**12**	**12**	**12**	N/S	N/S	**12**	**12**	**12**	**12**
04	315	**220**	**220**	273	**220**	**220**	220	**220**	220	220	253	220
05	**206**	**206**	**206**	**206**	**206**	**206**	N/S	N/S	**206**	**206**	**206**	**206**
06	372	**322**	**322**	368	**322**	**322**	368	**322**	326	326	322	344
07	518	N/A	N/A	518	N/A	N/A	370	N/A	213	213	349	227
08	282	**186**	**186**	277	**186**	**186**	252	**186**	200	188	260	251
09	733	436	436	733	436	436	630	436	436	436	491	500
10	609	244	N/A	568	N/A	N/A	484	**244**	369	263	386	301

Table 2: Results of the CP model (1)

ID	M13	M14	M15	M16	M17	M18
01	**14**	**14**	**14**	N/S	N/S	N/S
02	267	**226**	226	**226**	N/S	N/S
03	**12**	12	**12**	N/S	N/S	N/S
04	**220**	220	**220**	N/S	N/S	N/S
05	**206**	206	**206**	N/S	N/S	N/S
06	**322**	326	326	322	344	324
07	N/A	213	213	349	227	167
08	**186**	200	188	260	251	220
09	**436**	436	**436**	N/S	N/S	N/S
10	**244**	369	263	386	301	311

Table 3: Results of the CP model (2)

Inst. ID	M9	M10	M11	M12	M13	M15	M16	M17	M18	M19	final_config
11	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
12	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
13	544	560	1308	650	596	512	N/A	524	520	1146	484
14	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
15	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
16	440	N/A	1082	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
17	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
18	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
19	347	N/A	1465	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
20	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
21	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 4: Results of the CP model (3)

Inst. ID	**no_symm_br**	**max_load_symm_br**
01	**14**	*NO_SYMM*
02	226	226
03	**12**	*NO_SYMM*
04	220	220
05	206	*NO_SYMM*
06	322	322
07	261	243
08	186	186
09	436	436
10	244	244

Table 5: The results of the SAT model

ID	Z3
01	**14**
02	**226**
03	**12**
04	**220**
05	**206**
06	**322**
07	168
08	**186**
09	436
10	244
11	N/A
12	N/A
13	1322
14	N/A
15	N/A
16	1148
17	N/A
18	N/A
19	N/A
20	N/A
21	N/A

Table 6: SMT Experimental Result

ID	Gurobi NN Heuristic	Gurobi Baseline
01	**14**	**14**
02	**226**	**226**
03	**12**	**12**
04	**220**	**220**
05	**206**	**206**
06	**322**	**322**
07	167	167
08	**186**	**186**
09	**436**	**436**
10	**244**	**244**
11	305	N/A
12	347	N/A
13	530	518
14	435	N/A
15	398	N/A
16	286	286
17	658	N/A
18	327	N/A
19	334	N/A
20	N/A	N/A
21	376	N/A

Table 7: MIP Experimental Result