Combinatorial Decision Making and Optimization
Project Work: Multiple Couriers Planning

Benedetti Enrico
enrico.benedetti5@studio.unibo.it

Levita Luca
luca.levita@studio.unibo.it

Fantazzini Stefano
stefano.fantazzini@studio.unibo.it

Hartsuiker Jens Matthias
jens.hartsuiker@studio.unibo.it

Introduction

We are going to solve the Multiple Couriers Planning problem (also known as Capacitated Vehicle Routing problem), with different approaches: CP, SAT, SMT, MIP. Also provide comments, remarks and present experimental results.

Problem description

A set of n goods have to be delivered by a fleet of m vehicles to different customer locations. Each courier's vehicle i has a maximum capacity l_i. Also, each item j has a size s_j and needs to be transported to location d_j. It is required to plan a tour for each courier, i.e. define the sequence of locations visited to deliver the items. A courier's item load cannot exceed its maximum capacity. All couriers begin and end their route at a single depot, situated at the origin o. The objective is to minimize the total tour distance.

Instance format

For each instance there is a text file containing on each line different parameters of the problem.

- m: the number of couriers available;
- n: the number of customers to visit;
- the m capacities of each courier's vehicle;
- the n weights/sizes/quantities of goods for each customers' order;
- the x and y euclidean coordinates for the customers' and the depot/base positions.

Follows an example for the test instance given in the project pdf.

```
m = 3;
n = 7;
capacities = [15, 10, 7];
weights = [3, 2, 6, 8, 5, 4, 4];
Xs = [1, 2, 2, 4, 5, 5, 6, 3];
Ys = [3, 1, 5, 0, 2, 5, 4, 3];
```

CP

In the beginning, as for all three other approaches, we computed the distance matrix D (2). The distance matrix is generated by calculating the manhattan distance (1), also known as 1-norm or taxi distance, between each of the drop off points for each customer to visit.

$$\text{manhattan}(P_1, P_2) = |x_1 - x_2| + |y_1 - y_2| \tag{1}$$

$$D_{i,j} = \text{manhattan}(i, j) \tag{2}$$

Vocabulary

- A route is a succession of visits;
- The solution is made up of all the routes;
- Each route has a start and end point which coincide with the hub or depot;

Parameters

$COURIERS = \{1, \ldots, m\}$

the set of all couriers.

$PACKAGES = \{1, \ldots, n\}$

the set of customer locations.

array[0..n] of int: new_Xs
array[0..n] of int: new_Ys

Reorganization of the input coordinates having at index 0 the coordinates of the hub.

array[0..n] of int: $weights\_plus\_zero$

Reorganization of the input weights having at index 0 the placeholder weight for the hub, which is zero.

array[0..n] of int: $capacities\_sort$

Sorting the capacities of the courier in a descending order.

Decision variables

array[$PACKAGES$] of var $COURIERS$: $deliveredBy$

For each package i, $deliveredBy[i]$ define which courier operated the delivery for i.

array[$COURIERS$, 0..n] of var 0..n: $deliveries$

Matrix which defines for each courier the circuit of deliveries. Each row belongs to the relative courier while each column represent each customer, including the depot.

Constraints

The base constraint used to formulate the circuit is:

predicate subcircuit(array [int] of var int: x)

It constrains the elements of x to define a subcircuit where $x[i] = j$ means that j is the successor of i and $x[i] = i$ means that i is not in the circuit.

Below a simple example of our wanted behaviour hypotizing $m = 2$ and $n = 3$:

$$\text{courier1} = [1, 2, 0, 3]$$

which is equal to the route 0-1-2-0

$$\text{courier2} = [3, 1, 2, 0]$$

which is equal to the route 0-3-0

The constraint then applied to our model is:

$$\text{constraint forall(c in COURIERS) (subcircuit(row(deliveries, c)))} \tag{3}$$

The main advantage of using *subcircuit* instead of *circuit* is that it allows us to create the matrix with a fixed dimension since it doesn't require the presence of every index in the circuit.

To guarantee that each customer is served by one and only one courier:

$$\text{constraint forall(i in PACKAGES) (nvalue(col(deliveries,i)) = 2)} \tag{4}$$

Given a column i which represent a customer:

$$\exists\ i_x : i_x \neq i\ \forall y(i_y \neq i \rightarrow y = x);\ x, y \in COURIERS$$

The constraint (4) verifies that each column, except the one of the hub, has to have exactly two different values. This is done because we know that for every column only one row will have a value which will differ from the column index i, while the rest will all have i as value, meaning that the delivery for customer i is not performed by them.

The function *nvalue* in constraint (4) is used to return the number of distinct values in a given an array.

In order to prevent unwanted subcircuits for couriers which do not start nor end from the hub:

$$\text{forall}(c \text{ in } COURIERS \text{ where row}(deliveries, c)[0] == 0) \tag{5}$$

$$(\text{forall}(i \text{ in } PACKAGES)\ (\text{row}(deliveries, c)[i] == i)) \tag{6}$$

$$\forall c \in COURIERS,\ \forall i \in PACKAGES : deliveries[c, 0] = 0 \rightarrow deliveries[c, i] = i$$

Given all the rows where the courier doesn't departure from the hub (5), we know that every other customer cannot be visited by the afromentioned courier (6).

This constraint is checked by verifying that every column i for the courier c obtained from (5) has form:

$$deliveries[c, i] = i$$

Regarding *deliveredBy*, we have to check that the sum of the weights of the load for each courier c must not exceed its maximum capacity:

$$\text{sum}(i \text{ in } PACKAGES \text{ where } deliveredBy[i] = c)\ (weights\_plus\_zero[i]) <= capacities\_sort[c]) \tag{7}$$

Minizinc provides similar methods called Packing constraints such as the *bin_packing_capa* predicate which we actually used in our first approaches. We decided to discard them in this model because, after some testing, the performance were slightly better with the approach (7).

Also it's important to enforce the bidirectional relationship that *deliveredBy* and *deliveries* have (8) (9). Once the correct amount of packages has been loaded into a courier c we must check that the customer i is actually in the circuit performed by c.

$$\text{forall}(i \text{ in } PACKAGES \text{ where } deliveredBy[i] == c)\ (\text{row}(deliveries, c)[i] \neq i) \tag{8}$$

$$\text{forall}(i \text{ in } PACKAGES \text{ where } \text{row}(deliveries, c)[i] \neq i)\ (deliveredBy[i] = c) \tag{9}$$

The constraint (8) and (9) both produce the same result individually but adding redundant constraints, as we noticed here, improved the performance.

With just the previous constraints our model is now able to generate solutions but, for instances with numerous customers, the search space is too large to find good solutions. Good practice, in this case, is to implement other constraints which can limit the domain of all the possible decision variables.

The following symmetry breaking constraint prevents for each courier c mirrored circuits (e.g. [1,2,3]; [3,2,1]).

$$\text{forall}(i \text{ in } 0..n \text{ where } \text{row}(deliveries, c)[i] == 0)\ (\text{row}(deliveries, c)[0] > i) \tag{10}$$

$$deliveries[c, 0] > deliveries[c, LAST]$$

$$\text{where } LAST \text{ is the index } i \text{ that provides } deliveries[c, i] = 0$$

Knowing that for each courier c the first stop must be different, if it deliveres, we can infer that for two couriers c_1 and c_2:

$$\text{row}(deliveries, c_1)[0] \ \neq \ \text{row}(deliveries, c_2)[0] \tag{11}$$

$$deliveries[c_1, 0] \ \neq \ deliveries[c_2, 0]$$

Lastly, a constraint (12) which was proved empirically to be good, implies that its better to have the largest couriers filled up first:

$$\begin{gathered}\text{sum}(i \text{ in } PACKAGES \text{ where } deliveredBy[i] = c1)(weights\_plus\_zero[i]) \\ <= \\ \text{sum}(j \text{ in } PACKAGES \text{ where } deliveredBy[j] = c2)(weights\_plus\_zero[j])\end{gathered} \tag{12}$$

This last constraint, unfortunately, could prevent our model to find the optimal solution since it has a similar approach to the *min_courier* one, which we will later analyze in this paper. The main reason behind keeping (12) in our model was that, since this is a NP hard problem and a CP model applied to it can find exact solutions only for small instances, we opted to keep this last constraint since it largely improved the solutions;

Domain Search

To further improve the solution, search annotations in MiniZinc specify how to search in order to find, possibly, a better solution to the problem.

The final search annotation choosen are:

$$\text{int\_search}(deliveredBy,\ first\_fail,\ indomain\_split) \tag{13}$$

$$\text{restart\_luby}(1000) \tag{14}$$

This setup (13) was choosen after repeted testing, which showed that the best search was operated on *deliveredBy* using $first\_fail$ to start from the variable with the smallest doamin and *indomain_split* bisecting the domain and trying to exlude the upper half.

We also decided to insert a restart behaviour (14) since, for large domains, it can improve the search. The choosen restart behaviour was the *restart_luby* since it is largely used due to its empirical optimality.

Different approaches

Worth noting are some of our different approaches for the tour representation.

We asked ourselves if we could improve the representation of the circuits since it used a large space due to the utilization of a matrix and the presence of values that idealy could be omitted.

Initially, in order to avoid matrices, we opted for a vectorial representation of our circuit using the form:

$$\text{tour} = [0, 1, 2, 0, 3, 0]$$

Each courier was separated from another using 0, which represented the act of departing and returning to the starting point. This approach, given its semplicity, helped us analyzing all the possible contraints, but in terms of optimality it had large room for improvements (e.g. for inst01.dzn after five minutes it reached $totalDistance = 3324$).

This also helped us understand that not all the domain reduction constraints could coexist. An example is the intuition that the first customer of a tour should be closer to the origin than the second customer, but this assumption is not true if it coexists with the symmetry breaking

constraint (10). Another discarded, but worth mentioning, symmetry breaking constraint was the one concering the possibility of two couriers sharing the same circuit in different solutions. The final decision to discard this constraint was purely based on performance and result basis, since keeping it in the model reduced drastically the performances.

From this base, we implemented another array based representation that seemed to be largely shared among the literature. This approach is based on the utilization of two arrays: *succ* and *pred*.

The successor variables $succ_i$ gives the index of the direct successor of i while the predecessor variable $pred_i$ gives the index of the previous visit of i in the route. Even if the latter is redundant, emipirical evidence shows that it helps utilizing it too. Using the same example data used to explain constraint (3):

$$\text{succ} = [2, 6, 7, 1, 3, 4, 5]$$

$$\text{pred} = [4, 1, 5, 6, 7, 2, 3]$$

$$\text{Domain: } N \cup S \cup E$$

$$\text{Customers N} = \{1...n\}$$

$$\text{Start visits S} = \{n+1...n+m\}$$

$$\text{End visits E} = \{n+m+1...n+2m\}$$

Even with this slimmer representation, results did not improve the ones obtain with the matrix representation. The only improvement noticed where the computational speed on the samll test data instance and, sometimes, but not consistent, small improvement on the biggest instances like Inst06.

The last modification tried required the manipulation of the input data.
The first new input inserted was the *min_courier* data which represented the minimum amout of courier needed to actually complete all deliveries.

Even though this domain restriction actually improved some instances, this approach itself is not correct, since, as literature states, the minimization of the number of used circuits does not correspond, in general, to the minimization of the total cost of the circuits.

The second data inserted was a flattened matrix representing, for each customer i, the orderd sequence of the closest customer to i.

This approach actually speeded up the reasearch for the smaller instances like Inst01 where it reached small values such as $totalDistance = 1452$ but, unfortunately, did not manage to find any solution for larger instances.

Lastly, we considered implementing warm start just like we did in the MIP model since it largely improved performance but, constraint solvers do not support warm start.

At the end we reverted to the matrix representation which was the one with the better and consistent results. This can probably be explained by the simple but not simpler approach and the efficiency of the built-in predicate *subcircuit* in contrast to the wider, but in our case worse, *succ_pred* technique.

Results

The following table contains the best result obtained after repeted experiments with different combinations of constraints and search annotations on our model:

Instance	$totalDistance$
Inst01	2968
Inst02	6628
Inst03	13278
Inst04	16012
Inst05	18498
Inst06	25386
Inst07	5228
Inst08	12286
Inst09	19182
Inst10	24742
Inst11	3174

The results are far from optimal but they are generally better then the ones obtained using SAT and SMT, but still worse than the ones acquired using MIP.

References

[1] Models, relaxations and exact approaches for the capacitated vehicle routing problem, 123 (2002) 487 – 512 (Paolo Toth, Daniele Vigo)

[2] Handbook of Constraint Programming, Chapter 23: Vehicle Routing (F. Rossi, P. van Beek and T. Walsh)

[3] Constraint Programming for the Vehicle Routing Problem, 19th International Conference on Principles and Practice of Constraint (Philip Kilby)