Pizza Guy

Giacomo Boldini, Alessio Diana, Federica Zaglio

February 14, 2022

Abstract

In this report, we will describe the Pizza Guy project. This project aims to find the best possible schedule of a set of orders that minimizes the distance traveled by the deliverers in a evening of work. Our solution is developed using a CLP(FD) model and implemented in MiniZinc. All the remaing part of the project (data retrieving, cleaning and visualization) are done using Python.
In the section 1 we will describe the problem and the goal we want to reach, in the section 2 we will the describe the model and the implementation, including some euristichs and symmetry breaking strategies. In the section 3 we will describe the results and the possible search strategies. Finally, in section 4 we will conclude that the main problem in the proposed model is the solving time, that quickly increases considering a number of orders greater than 10 and a number of deliverers greater than 4, which are relatively small. Also, some possible changes and improvements will be proposed.

Contents

1 The problem

The main problem we want to solve can be described as follow: the best pizzeria in the town delivers pizzas at home; we want to find the best schedule of the set of orders (for a single evening) that minimize the total traveled distance by the deliverers respecting the delivery times requested by the customers.

The assignment gives us the following informations about the problem:

1. an order consists of:
   (a) a delivery address;
   (b) a desired delivery time;
   (c) a number of pizzas;
2. the desired delivery time has a granularity of 15 minutes (from 19.00 to 22.00);
3. the delivery window is up to 30 minutes later than desired delivery time;
4. the street topology of the city can be seen as a graph: the nodes represent the addresses and they are connected by edges that contain travel time between nodes; so, we know the distance (in terms of time) between each pair of addresses in the town;
5. every deliverer can carry at most 16 pizzas;
6. multiple travels from and to the pizzeria are allowed even in the delivery window.

To solve the problem means to assign a set of sorted lists of orders to every deliverer. Each list represents a travel and all the orders in it must be handled (or, in other words, the destination of this order must be reached) in the specified sequence.

1.1 Assignment

The goals of this work are:

1. write a program that solves the problem (in MiniZinc);
2. write a benchmark suite using 5 different towns of increasing size and 10 input sets for each one; the input sets must enforce that all the deliverers do at least two travels (in section 3.2)
3. on one configuration that runs in a couple of minutes, try different search strategies and then use the most promising one to solve a difficult input set (in section 3.1).

2 Model

2.1 Input data

The input consists in having:

1. d: the number of deliverers available
2. $mdist$: the 2-d matrix representing the travel distance between every pair of nodes in the graph generated using Dijkstra (including those that are not a destination)
3. k: the side dimension of $mdist$
4. a set of N orders, each of them made by a destination $dest$, a delivery time $orario$ and a number of pizzas num_pizzas

$$\begin{aligned} \text{orario} &= [o_1, o_2, \ldots, o_N]; \\ \text{num\_pizzas} &= [np_1, np_2, \ldots, np_N]; \\ \text{dest} &= [d_1, d_2, \ldots, d_N]; \end{aligned}$$

2.2 Assumptions

In order to make the problem a little easier, we made the following semplifications:

1. the graph is undirected, so `mdist` is symmetric:

   ```
   mdist[x, y] = mdist[y, x]
   ```

   As we'll show later, we can use Dijkstra on a real-world graph to calculate distances between the nodes. In this case, `mdist` matrix could be asymmetric.

2. a deliverer can do only a single travel in one half-an-hour. In other words, the orders that are assigned to him/her are delivered creating a circuit starting from the pizzeria, visiting all the destinations and ending in the pizzeria in one half-an-hour;
3. distances and travels time are considered the same thing;
4. the set of orders is known before the start of the evening;
5. all the time needed by the deliverer to interacts with customers (give them the pizzas, eventually handle the payment and other stuff) is supposed to be zero.

2.3 Implementation

The time is divided in 6 slots of half-an-hour (19.00 - 22.00). Let's call h one of these slots of time. There are d deliverers available for doing deliveries. Each deliverer, d, has a set of orders to handle for each slot of time h.

Scheduling These informations are stored in a boolean 3-d matrix called `scheduling` in which:

`scheduling[`d`, orderID, `h`] = true`

means that the deliverer d has to deliver the order with id `orderID` in the time slot h. In every slot h, the deliverer must reach all nodes specified as *destinations* (`dest`) of the orders assigned to him/her, starting from node 1 (pizzeria) and returning back to it. This creates a sort of *circuit* starting and ending with the same node, that reaches all the nodes specified in the matrix row.

Obviously, one order can be delivered only once during the evening by only one deliverer, and must be done in the delivery window computed from the requested time.

```
array[1..d, 1..N, 1..h] of var bool:  scheduling;
```

Pizzas_carried During the evening there are N orders to be delivered, each of which is made of 16 pizzas at most. This ensure that we need only a deliverer to deliver the entire order. Since we can deliver more than one order in a single travel, we need to record the total number of pizzas that a deliverer has in his/her bag. To do this, we use a 2-d matrix called `pizzas_carried` where the first dimension represents the deliverer d and the second represents the half-an-hour h considered.

```
array[1..d, 1..h] of var 0..16:  pizzas_carried;
```

Ea - Estimated arrival In order to ensure that all deliveries arrive in time, we use `ea` which stands for estimated arrival. This array contains, for every order, the arrival time (in minutes) from the starting time (19.00). At every order, we sum the number of minutes that must pass before the delivery take place, so doing this way, we are sure that the delivery is done in the right delivery window. For example, if the delivery is requested for 20.00, if the destination is five minutes far from the pizzeria and no other delivery is made before that, we will sum 60 to be sure about the right delivery window and 5 for the travel time. So the `ea` for this delivery will be 65.

```
array[1..N] of var 0..h*30:  ea;
```

P The main goal of the project is to minimize the travel distance covered by all the deliverers in the evening. To do so, a possible strategy is to deliver more than one delivery in a single travel. This is convenient if the deliveries are made in a way that minimize the distance traveled between the nodes. To encode this information, we use `P`, which is a 3-d matrix that stores all the nodes that every deliverer has to visit in one travel, according to `scheduling`.

```
array[1..d, 1..N, 1..h] of var array2set(dest) union 0:  P;
```

X This 3-d matrix is used to support the creation of all the other constraints, making the computation of the distances easier. In this matrix, meangingful *positions* of the matrix P are stored for each d and h. Each row contains the positions of the matrix P (destination nodes) that d deliverer has to reach in the h half-an-hour. This matrix allows us to control the order of each delivery made by every deliverer in one travel and, as consequence, compute the corresponding travel distances (stored in `distances`).

```
array[1..d, 1..N, 1..h] of var 0..N: X;
```

Distances Using the data structures seen before, we use this 3-d matrix to store all the partial distances needed to reach every delivery node starting from 0 for each half-an-hour. Using this matrix we can enforce that all the deliveries are delivered in time and, since we want to minimize the traveled distance, it allows us to combine deliveries that belongs to different time slots (eg. 19.15 and 19.30) but only if they are adjacent because of the maximum delivery time of 30 minutes.

```
array[1..d, 1..N, 1..h] of var 0..29:  distances;
```

The matrices *scheduling*, P and *distances* are strictly related to each other. They encode the same concept but using different informations. First of all they have the same sizes `[1..d,1..N,1..h]` and if one cell is meaningful in one of them, it is also meaningful in the others. In particular, in a position `[id,iN,ih]`:

- *sheduling* = *1* means that the order *iN* has to be accomplished by the deliverer `id` in the `ih` half-an-hour;
- $P = n$ means that the order *iN* has to be accomplished by the deliverer `id` in the `ih` half-an-hour reaching the node `n` in the graph;
- *distances* = l means that the order `iN` has to be accomplished by the deliverer `id` in the `ih` half-an-hour and it will take l minutes to reach it, starting from the beginning of the corresponding half-an-hour (partial time).

All the others meaningless positions are set to zero in all the matrices.

Total_travel We use this 2-d matrix in order to store the total travel distance of deliverer *id* in one half-an-hour *ih*. As described above, in the matrix *distance* there are partial distances: from pizzeria for each delivery node handled, starting from the corresponding half-an-hour. If we add the distance between the last delivery node and the pizzeria (that is the path to return to the pizzeria) we have the total travel distance. As already mentioned, our goal is minimize the sum of all these values.

```
array[1..d, 1..h] of var 0..29:  total_travel;
```

2.4 Constraints

To ensure `scheduling` consistency, the following constraints are made:

- `sum(scheduling) = N;` ensures that all the orders are fulfilled
- `forall(j in 1..N)(`
  `sum([scheduling[id,j,ih]| id in 1..d, ih in 1..h]) = 1`
  `);` ensures that only one delivever takes care of one order.

To construct the matrix `P` a simple product between a `scheduling` row and the array of the destinations `dest` is made:

```
forall(id in 1..d, iN in 1..N, ih in hra[iN])(
   P[id, iN, ih] = scheduling[id, iN, ih] * dest[iN]
);
```

Thanks to this product, we are able to make meangingful the same positions of `scheduling`.

The `X` matrix has to contain all the meaningful positions in the matrix `P`. To ensure this, we need a simple way to sequentially compute all the informations related in a single row. For every value greater than zero found in a row of `P`, we enforce the corresponding row of `X` to contain it. Also, we want to push all the meaningless values (zeros) to the end of the row: this is done by another constraint described below (section 2.4.1 (3)).

```
forall(id in 1..d, iN in 1..N, ih in 1..h)(
   if P[id, iN, ih] > 0 then
      count([ X[id, iNN, ih] | iNN in 1..N], iN, 1)
   else
   (
      count([ X[id, iNN, ih] | iNN in 1..N], iN, 0)
      /\
      distances[id,iN,ih] = 0
   )
   endif
);
```

To compute distances means to fill the `distances` matrix. To do this, we have to use the matrix `X`. Considering a single row, we must take care of the following cases:

- the first (meaningful) cell;
- the other (meaningful) cells.

In the first case, if `X[1]` is the first meaningful position in the matrix P and `P[X[1]]` is the first visited node in a travel, the distance of the node `P[X[1]]` is computed summing the distance between the pizzeria (node number 1) and this node: `mdist[1, P[X[1]]]`. This value is put in `distances[X[1]]`.

In the second case, the distance to reach the node in `X[i]` is the sum of the distance between the node `X[i]` and `X[i-1]`, and the last distance previusly computed (`distances[X[i-1]]`).

For the sake of simplicity, in the example above, the matrices P and X are treated like arrays to avoid writing all the other indexes that are fixed because we are considering a single row.

```
forall(id in 1..d, iN in 1..N, ih in 1..h)(
   if (X[id,iN,ih] !=0  /\ P[id, X[id,iN,ih], ih] != 0) then
      if iN == 1 then
         distances[id, X[id,iN,ih], ih] =
            mdist[1, P[id, X[id,iN,ih], ih]]
       else
          distances[id, X[id,iN,ih], ih] =
            mdist[P[id, X[id,iN-1,ih], ih], P[id, X[id,iN,ih], ih]]
          + distances[id,X[id,iN-1,ih], ih]
                  | iNN in 1..iN-1])
       endif
   endif
);
```

Moreover, using `X`, we must check the "travel consistency", which means that the deliverer must be able to return back to the pizzeria before the start of the next half-an-hour. To do this, we store the computed travel distance in `total_travel` and its domain `[0..29]` ensures this property.

```
forall(id in 1..d, iN in 1..N, ih in 1..h)(
  ( X[id,iN,ih] !=0
    /\ ( iN+1 == N+1 \/ X[id,iN+1,ih] == 0)
    /\ P[id,X[id,iN,ih],ih] != 0)
        -> (total_travel[id,ih] =
              distances[id, X[id,iN,ih], ih]
              + mdist[P[id, X[id,iN,ih], ih],1])
);
```

To make `ea` consistent, which is the array that allow us to record the expected arrival time of each delivery, we take into account the row representing the delivery in the `scheduling` matrix and the same position in the `distances` matrix. Since `scheduling` is a boolean matrix, only the meaningful position in `distances` will be used. For every half-an-hour from the starting time to the beginning of the corresponding half-an-hour, 30 is added to this value. In addition, we must constraint that every delivery is delivered within 30 minutes starting from the desired delivery time.

```
forall(iN in 1..N)(
   ea[iN] = sum([scheduling[id,iN,ih]
            * (distances[id,iN,ih]+((ih-1)*30))
               | id in 1..d, ih in 1..h])
   /\
   ea[iN] >= ra[iN]
   /\
   ea[iN] < ra[iN]+30
);
```

The consistency of `pizzas_carried` is trivial. For every order, we enforce that the number of pizzas carried by a deliverer in one half-an-hour is the same as the number of pizzas in the orders that he/she is taking care of in the same half-an-hour.

```
forall(id in 1..d, ih in 1..h)(
   pizzas_carried[id, ih] = sum([ num_pizze[j]
                                | j in 1..N
                                where scheduling[id, j, ih] = 1])
);
```

Some of the data structures previously analized could have been avoided if we wanted to have a more lightweight model. We introduced them to make simpler the creation of the constraint and to keep the model readable.

2.4.1 Restrict the search space

Creating the model, we included some easy euristichs that can restrict the dimension of the search space:

1. domain restrictions.

   As said in section 2, the domain for each decision variable has been strictly restriced to the only possible values that they can assume. For example, a cell of `distances`, that represent the partial travel time to reach a destination, can't be greater than 29 since we want that each deliverer returns back to the pizzeria before the next half-an-hour. Another example is the domain of `P`: in this case we restrict the domain from `0..k` (which

is correct) to only the set of possible destinations. This reduces a lot the number of possibility in the domain.

2. for each order, exclude all the non-possible half-an-hour.

   Every order has a requested delivery time that becomes a delivery window. We know that all the half-an-hours before and after this delivery window are not usable. So, for each order we computed its *valid* half-an-hour indeces (`hra`) that are used to access the matrices `scheduling`, `P` and `distances` in *not-valid* half-an-hour to set it equal to zero. Also, a consideration on `X` matrix is made: for each *not-valid* half-an-hour for a specific order, we know that this order index will be not present in the corresponding row of `X`.

   ```
   forall(id in 1..d, iN in 1..N, ih in (1..h diff hra[iN]) ) (
      scheduling[id,iN,ih] = 0
      /\
      P[id,iN,ih] = 0
      /\
      distances[id,iN,ih] = 0
      /\
      count( [X[id,iNN,ih] | iNN in 1..N], iN, 0)
   );
   ```

3. enforce useless cells of matrix `X` to zero.

   Each line of `X` contains the indexes of the corresponding line of `P` that are meangingful (different from zero). Also, all the zero in this (`X`) line must be pushed to the end of it. Knowning this, if we count the number of zeros in each line of `P` we can contraint the same number of cells to be zero, starting from the end of line of `X`.

   ```
   forall(id in 1..d, ih in 1..h)(
      let {
         var int: c = count([P[id,iNN,ih] | iNN in 1..N],0)
      }
      in forall(iN in N-c+1..N)(
         X[id,iN,ih] = 0
      ) /\
      forall(iN in 1..N-c)(
         X[id,iN,ih] > 0
      )
   );
   ```

2.4.2 Simmetry breaking strategy

Having more than one deliverer implies the possibility of having the same solution multiple times. For example, if we have three delivery and two deliverers,

known that the best strategy is to deliver the first two togheter and the third alone, one solution is to give the first two to the deliverer `d1` and the third to the deliverer `d2`. A simmetric and equivalent solution is to give the first two to `d2` and the third to `d1`.

In order to avoid this kind of problem, we impose an ordering on the deliverers by counting the number of pizzas that are carried in one half-an-hour. This enforce that the first deliverer carries a number of pizzas greater or equal to the number of pizzas carried by the second deliverer, the second deliverer a number of pizzas greater or equal to the number of pizzas of the third deliverer and so on.

```
forall(ih in 1..h, id in 1..d-1)(
   pizzas_carried[id,ih] >= pizzas_carried[id+1,ih]
);
```

3 Results

In this section, we will describe how the model behave, mainly in terms of solving time, using different input data and different search strategies.

To do this, we consider 5 italian city of increasing size (listed by increasing dimension): *Visano(BS)*, *Asola(MN)*, *Montichiari(BS)*, *Brescia (BS)* and *Roma (Roma)*. Their street graphs have the following characteristics:

City	nodes	edges
Visano	297	378 to
Asola	480	664
Montichiari	1130	1509
Brescia	3925	5464
Roma	4729	7277

As preprocessing step, for each of them we computed the shortest path beetween all the pairs of nodes using Dijkstra. This produced the corresponding $mdist \in M^{nodes \times nodes}$.

In addition to the model tests, we report a *real-world* result of a model execution, obtained using some pre- and post-processing techniques.

3.1 Search strategies comparison

Choosen a configuration of the problem that runs in a couple of minutes, we tested how it performs (in terms of solving time) changing the search strategies. The configuration choosen is:

$$N = 12 \text{ (orders)}, d = 2 \text{ (deliverers)}, \text{city} = \text{Visano}.$$

First of all we made a preliminary analysis, which results are shown in section 3.1.1. Then, more specific analyses are made on a subset of possible search strategies (the most promising among all) and results are shown in section 3.1.2.

The best strategy we found in the preliminary analysis (in terms of solving time) is the one shown in table 1. However, as we found later, searching on variable `P` isn't the optimal one (on average). Indeed, later we used the `scheduling` variable, which turned out to be the best one (on average).

var	P
vsa	anti_first_fail
vca	indomain
initTime	0.665783
solveTime	4.45827
solutions	2.0
variables	5925.0
propagators	2322.0
propagations	2118705.0
nodes	5179.0
failures	2916.0
restarts	0.0
peakDepth	18.0

Table 1: Search strategy with minimum solve time in preliminary analysis.

3.1.1 Preliminary analysis

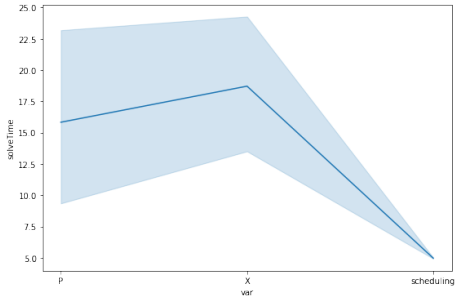
First of all, we made preliminary analyses to figure out the most promising search strategies among all their possibile configurations. To do this, we simply analyzed the variation of solving time changing the search annotations in Minizinc: in particular, we changed (1) the search variable, (2) how the variable is choosen and (3) how to constraint the variable.
Results are shown in fig. 1 and we can see that searching of variable `scheduling` could be the faster choice. Also, some VCA (Variable Choice Annotation) and VSA (Variable Search Annotation) appear to be more promising than others.

3.1.2 More specific analysis

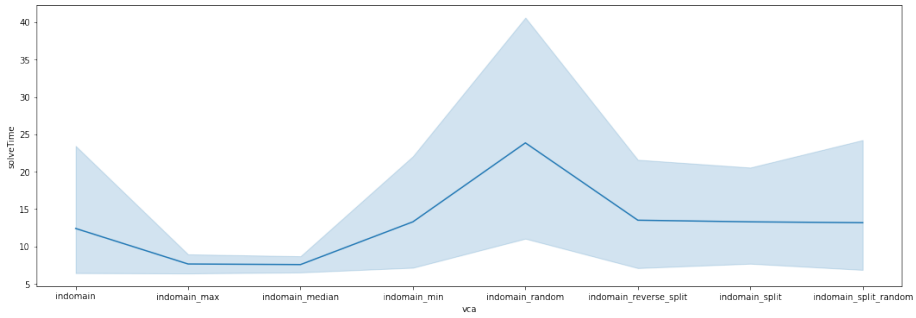
In this case, we selected a subset of search strategies (the most promising) and then we made a more specific analysis, reducing the timeout to 20 seconds and testing the same strategy more than one single time. All the executions that reached the timeout are removed from the analysis.

We computed the correration matrix (fig. 2) between search statistics and it shows high correration between:

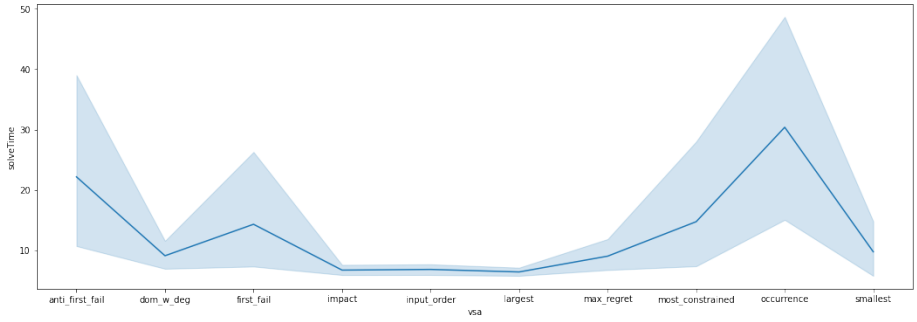
- number of failures and number of nodes (1);
- number of propagations and solve time (0.79): this, in addition to the non-correration between maximum depth and solve time (-0.05), points out that as the number of propagations increases, it also increases the solve time, but not the tree's depth.



(a) changing search variable.



(b) changing Variable Choice Annotation (VCA)



(c) changing Variable Search Annotations (VSA)

Figure 1: How solving time behave (on average) changing search strategies configurations.

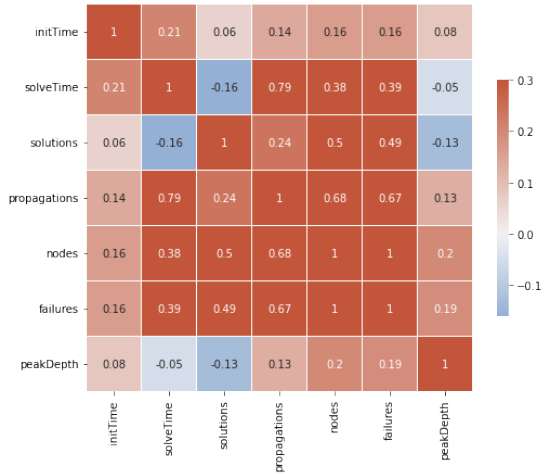


Figure 2: Correlation matrix between the search statistics.

var	count	mean	std	min	25%	50%	75%	max
scheduling	280.0	5.3998	0.5834	4.7762	4.87248	5.4201	5.576	8.7637
X	240.0	9.2220	2.7396	5.8241	7.03597	8.7616	10.510	20.0438
P	251.0	9.4433	5.0247	4.9271	5.55710	6.4068	15.243	19.5353

Table 2: Solve time statistics

In fig. 3 more detailed solve time analyses are reported, for each search variable. It's clear that:

- regardless of which VCA and VSA are choosen, searching on `scheduling` variable is faster than searching on `X` or `P` variables (again, on average).
- searching on `scheduling` variable never leads to timeout executions.

The table 2 shows a summary of these informations.

3.2 5 city of increasing size

Using one of the best search strategies, we tried to solve the problem using all the 5 increasing size cities in order to see how solving time changes. The timeout is set to 5 minutes for each execution. In this case, the configuration choosen is:

$$N = 12 \text{ (orders)}, d = 2 \text{ (deliverers)}$$
$$\text{city} = \text{Visano, Asola, Montichiari, Brescia, Roma}$$

Solving times found are shown in fig. 4. Solve and init times are set to 300 seconds if the execution reached the timeout without solving the problem. This is the case for the two biggest cities (Brescia and Roma). For the other cities, the time increases with the size of the city and it is always under 20 seconds.

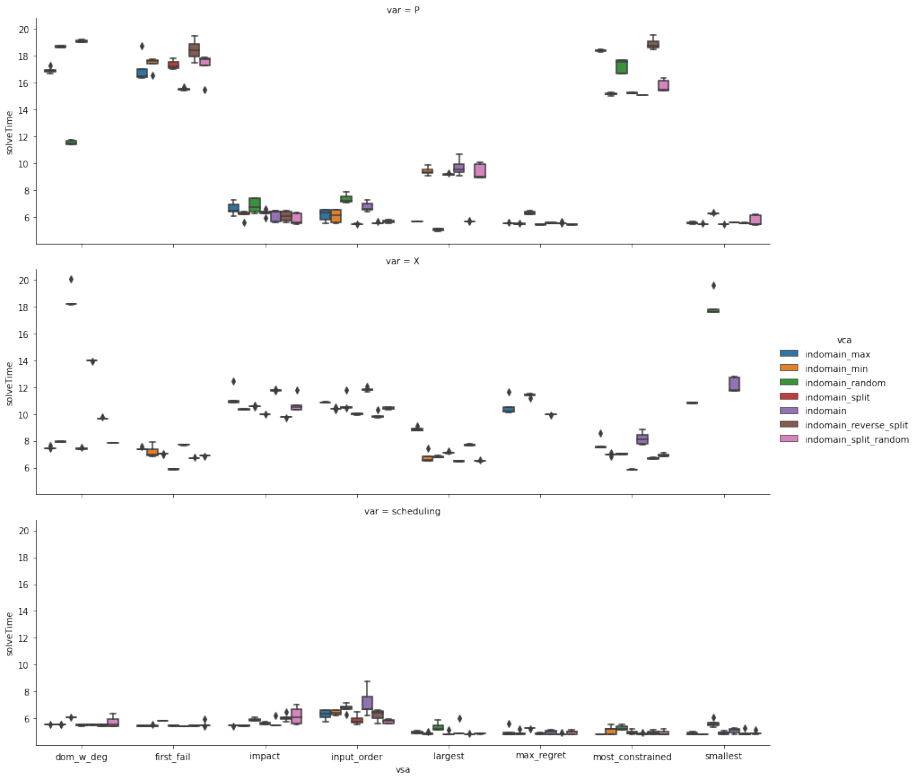
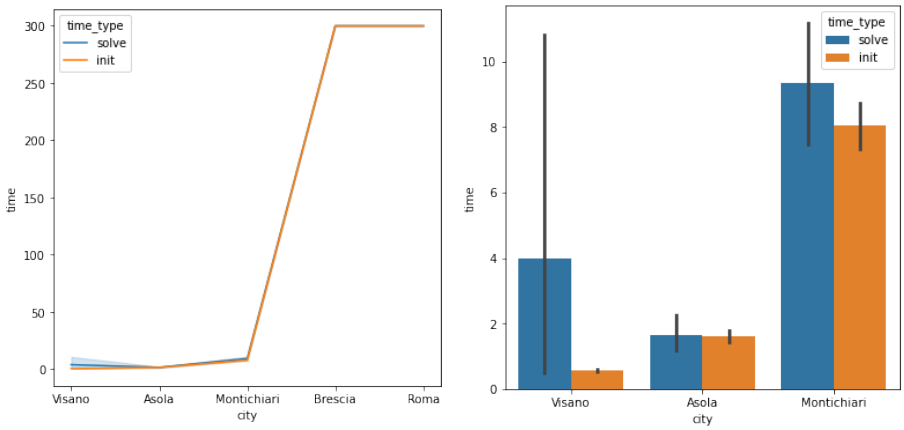


Figure 3: More specific analysis on solve time, changing the search strategies.



(a) (b)

Figure 4: Solving times for 5 cities of increasing size. $N = 12$, $d = 2$

3.3 Increasing the size of the problem

Using one of the best search strategies, we tried to solve a set of problems in the same city (Visano) with different complexities, represented by number of order N and number of deliverer d. The timeout is set to 2 minutes for each execution, N ranges from 2 to 15 and d ranges from 1 to 4. Results are represented in fig. 5 and they show what we expected: the peak depth of the tree, the solve time, the number of solutions and propagations increases with N and d.

3.4 Real-world application

Using *Python* and *OpenStreetMap* we were able to execute the model on a graph obtained by a real-world map and plot the results directly on the streetmap.

In fig. 6 there is an example of model execution on the city *Visano*, with $N = 13$ orders and $d = 2$ deliverer. Solve time was 48.8 seconds.

4 Conclusions

Our proposed model seems to be working, *but* the main problem is the solving time: it increases quickly with the size of the problem. Indeed, the model takes a couple of seconds to solve the problem only with a number of order $N \leq 12$. Using a number of orders over this threshold, the executions always reach the timeout of 5 minutes. With such a small value of N, we're far from a real world situation, which usually includes $N \geq 50$ (and more than $d = 3$ deliverer).

Also, we tried to use different solvers (from Gecode), but we got no improvement; on the contrary, we obtained worst results.

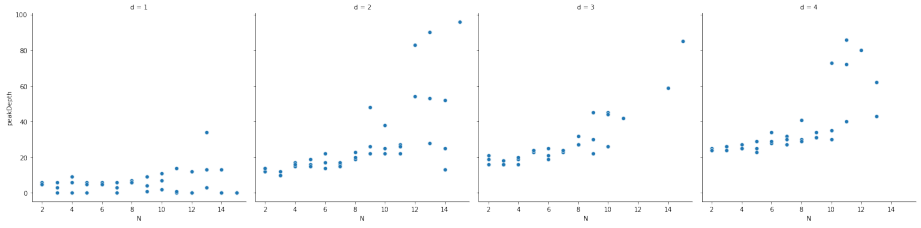
One possible improvement that we tried is reducing `mdist` size, considering only the destination nodes. This requires some pre- and post-processing steps and removes the correlation between the size of the city (number of nodes) and the time required to solve the problem; However, considering the complexity (N and d) the required time keeps growing in the same way.

Future works

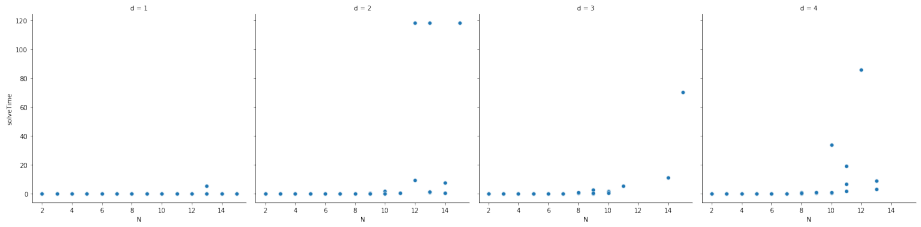
First of all, rethink about the model could be useful in order to reduce the solve time; maybe a simpler model leads to lower times. This can be done in parallel with `mdist` reduction, which is still a good choice. Also, it can be a good idea to find a way to exclude from the same travel those orders whose destinations are too far away to be made in a single travel.

Talking about the goal, only total traveled distance is minimized (for now); maybe a balancing function between the deliveres could be useful.

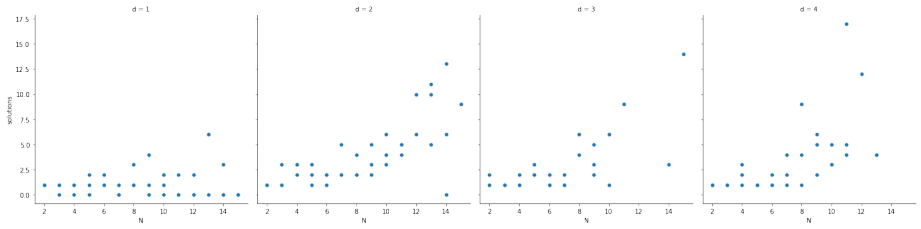
Finally, for a more realistic application, take care of the time needed by a deliverer to give the pizza to the customer, eventually handle the payment and other stuff could be useful.



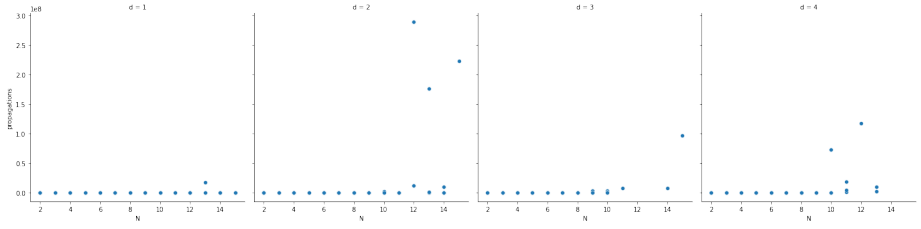
(a) peakDepth



(b) Solve time

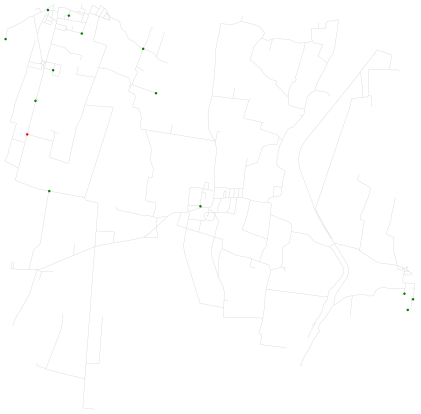


(c) Solutions



(d) Propagations

Figure 5: Results increasing the size of the problem. $N = 2 \dots 15$, $d = 1 \dots 4$, city = Visano. 2 minutes timeout.



(a) Pizzeria (red) and destinations to reach (green)

(b) Path for each deliverer

Figure 6: Map plot of model's results.