Algorithms for Intelligent Decision Making (CS4210-A) TU Delft – Spring 2020 Report for Part 1 Group number 28

Thalis Papakyriakou - 5007380 Andrea Alfieri - 5128315

2nd March 2020

Part 1 ROADEF'05 challenge

All experiments were run under Windows 10 Education (64 bit) on an I7-4770K processor running at 4.2GHZ, 16GB RAM running at 3200MHZ and a GTX1080TI GPU.

A Overview of model formulation - Viewpoints

Our model formulation came from questioning which would be the most efficient way of deciding the 3 variables of the optimization function: the number of high-priority constraint violations, the number of low-priority constraint violations, and the number of paint changes. We concluded that, if we had an array indicating the order in which the vehicles were scheduled, we could iterate over that array and count, per constraint, how many violations there were, and how many paint changes there were. The data structures already given to us in the skeleton made every lookup simple: for every vehicle we went through, all their data attributes were easily accessible, and setting the decision variables was only a matter of correctly calculating the violations. The rules regarding the number of violations were quite specific, and so, the simplest the model, the easier it would be for us to correctly and coherently define those rules in our model.

No redundant variables were used; every piece of information was easily accessible through simple data structures. No implicit constraints were used; every constraint was explicitly and clearly defined through the use of the data structures at hand. Symmetry-breaking constraints and inference annotations were used. Search strategies were not used as none proved better than the default one. More will explained further on in the report.

We did not formulate a secondary model / develop a new viewpoint. We tried thinking of a less straight-forward solution that would actually help the solvers/backend more, but in the end nothing came up.

As we will try to explain further on, we consider our current model clear, simple, compact (few variables/constraints), with easy value look-ups, and with a satisfying performance.

B Model

In this section we provide an extensive discussion about the model described by the code, which can be found below. The parameters provided in the skeleton code will not be discussed here, they are `paintBatchLimit`, `numVehicles`, `ratios`, `associatedConstraints`, `vehicles` and `objectiveMultipliers`.

B.1 Variables

There are many variables that we added to the skeleton code, the most important being the **schedule** decision variable. This variable is an array of size *numVehicles* which represents the order in which the vehicles will be scheduled. Each vehicle is defined by an ID, which is the position of the vehicle in the input data. Therefore, `schedule[5] = 3` means that the vehicle with ID $= 3$ will be served 5th. Because this array contains the solution to the problem, we will also refer to it as the **solution** array. Our model contains 7 constraints, of which 5 refer to the schedule array.

The other 2 constraints are only used to define the decision variables **highViolations** and **lowViolations**, which represent the number of high-priority and low-priority violations of each solution the solver is able to find. This is done through the use of a third support variable called **violations**: an array of size *Constraints* where `violations[5] = 3` means that, in the found solution, there are 3 violations of the 5th constraint type.

numPreviousDayVehicles and **numOfColours** are two more support, non-decision variables that are used respectively to count how many vehicles come from the previous day (and have therefore a pre-fixed position in the schedule array) and how many different colors are found in the input data.

At last, **paintChanges** is the decision variable used to count how many times the factory has to change paint, which is determined by counting how many times two vehicles of different colors are found next to each other in the schedule array.

B.2 Constraints

The constraints at line 65 and 71 are used to define the support variables **highViolations** and **lowViolations** as

$$\text{highViolations} = \sum_{c \in H} \text{violations}[c]$$

$$\text{lowViolations} = \sum_{c \in L} \text{violations}[c]$$

where H and L are the sets of high and low priority constraints.

The constraint at line 77 completely defines the **paintChanges** variable by counting how many paint changes occur in the solution array:

$$\text{paintChanges} = \sum_{i=\text{numPreviousDayVehicles}+1}^{\text{numVehicles}-1} f(i)$$

$$f(i) = \begin{cases} 1 \text{ if } \textit{color of vehicle}\, i \neq \textit{color of vehicle}\, (i+1) \\ \text{else } 0 \end{cases}$$

The constraint at line 93 just states that all values in the **solution** array need to be distinct. The constraint at line 97 states that if a vehicle comes from the previous day, its position in the **schedule** array will be the same as the one in the input data:

$$\forall i \in [1; \text{numVehicles}] : \text{vehicles}[\, i,\, \text{Date}\,] = 0 \Rightarrow \text{schedule}[\, i\,] = i$$

The constraint at line 105 is used to ensure that there are no sequences of vehicles of the same paint color in the schedule that are longer than allowed by **paintBatchLimit**. However, as a preprocessing measure, the constraint is only applied if there is at least one colour, for which the number of vehicles to be painted with it is more than the paintBatchLimit. For this constraint, we use the MiniZinc function *nvalue*, which returns the number of distinct values in a given array.

The constraint at line 114 has been the most difficult one to write, as it is used to count the amount of violations of each soft constraint type. We first define a *window* as a set of vehicles which are next to each other in the solution array. The size of the window changes for each soft constraint type, as it is determined by the *Denominator*. For each constraint type, we move the window along the **schedule** array and count the amount of violations of that constraint for that window through the given formula:

$$\left(\sum_{j \in \text{window}} \text{associatedConstraints}[\,\text{schedule}[\, j\,],\text{c}\,] \right) - \text{ratios}[\,\text{c}, \text{Numerator}\,]$$

Since this function returns a negative value when the window respects the given constraint, we use the $max(0, x)$ function in order to return the right amount of violations, and use it to count the overall sum of violations.

The constraint at line 143 is the only symmetry breaking constraint of our model and our tests have been run with and without this constraint in order to test its efficiency. This just states that if two vehicles have the same color and the same constraints, the one with the bigger ID has to come first in the schedule array.

Listing 1: Our MiniZinc model for the assignment

```
1  include "globals.mzn";
2
3  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% PARAMETERS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
4
5  % The different properties of each ratio constraint.
6  % These properties are instantiated in the "ratios" parameter variable.
7  % - Numerator:   the numerator of the provided ratio
8  % - Denominator: the denominator of the provided ratio
9  % - Priority:    whether the constraint is high-priority (1) or not (0)
10 enum RatioProperties = {Numerator, Denominator, Priority};
11
12 % The different properties of each vehicle.
13 % These properties are instantiated in the "vehicles" parameter variable.
14 % - Date:        whether the vehicle is from the previous day (0) or not (1)
15 % - PaintColour: which colour of paint the vehicle has
16 enum VehicleProperties = {Date, PaintColour};
```

```

% The different objectives within the situation.
% - HPRatioConstraints: the objective regarding the ratio
  constraints with Priority == 1.
% - LPRatioConstraints: the objective regarding the ratio
  constraints with Priority == 0.
% - PaintColourBatches: the objective regarding the number of paint
  changes.
enum Objectives = {HPRatioConstraints, LPRatioConstraints,
  PaintColourBatches};

% The upper limit on the batch size of paint colour (how many
  consequetive same-colour vehicles are allowed).
int: paintBatchLimit;

% Number of vehicles within the scenario.
int: numVehicles;

% The names of the available ratio constraints.
enum Constraints;

% The properties of each available ratio constraint.
array[Constraints,  RatioProperties] of int: ratios;

% The associated ratio constraints for each vehicle.
% Each cell corresponds to whether a vehicle is associated with a
  constraint (1) or not (0).
array[1..numVehicles, Constraints] of 0..1: associatedConstraints;

% The associated other properties for each vehicle, as described in
  "VehicleProperties".
% Note that the ordering of the vehicles of the original datafiles
  are preserved.
% Therefore, the schedule of the previous day (the vehicles with
  "Date" == 0) can be derived
% from the ordering within this matrix.
array[1..numVehicles, VehicleProperties] of int: vehicles;

% The multipliers of each available objective, which are described
  in "Objectives."
% In the original datafile, the first-ranked objective receives a
  multiplier of 10000,
% the second a multiplier of 100 and the third a multiplier of 1.
  If an objective is
% not included in the optimization objectives, its multiplier is 0.
array[Objectives] of int: objectiveMultipliers;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
  DECISION VARIABLES
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% The vehicles in the order they will be processed. This is our main, key decision variable
% whose values will dictate the values of all the others. We will be using its name, "schedule", often.
array[1..numVehicles] of var 1..numVehicles: schedule;

% An array that describes the number of violations per constraint in our schedule.
array[Constraints] of var int: violations;

% Total number of violations of the high-priority constraints.
% The associated constraint is basically defining the decision variable, summing
% up all the different high-priority constraint violations counted in the schedule.
var int : highViolations :: is_defined_var;
constraint highViolations = sum(c in Constraints where ratios[c,Priority] = 1)(violations[c]) :: defines_var(highViolations);

% Total number of violations of the low-priority constraints.
% The associated constraint is basically defining the decision variable, summing
% up all the different low-priority constraint violations counted in the schedule.
var int : lowViolations :: is_defined_var;
constraint lowViolations = sum(c in Constraints where ratios[c,Priority] = 0)(violations[c]) :: defines_var(lowViolations);

% The variable represents how many times the paint has been changed in the schedule,
% and is defined by the constraint counting the number of different-colour pairs of vehicles
% in the schedule.
var int : paintChanges :: is_defined_var;
constraint paintChanges = count(
  [ vehicles[schedule[i], PaintColour] != vehicles[schedule[i+1],PaintColour]   | i in numPreviousDayVehicles+1..numVehicles-1],
    true
) :: defines_var(paintChanges);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SUPPORT VARIABLES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

```
% Support, non-decision variables.
int : numPreviousDayVehicles = count(vehicles[..,Date],0);   % Number of vehicles from the previous day.
int : numOfColours = max(vehicles[..,PaintColour]); % Number of different colours.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% CONSTRAINTS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Ensures that no vehicle ID is used twice within the schedule.
% Here, domain consistency rules out an option for every other cell in the    schedule    array
% every time a cell is assigned a value. Helpful, and not significantly negatively impactful on the runtime.
constraint all_different(schedule) :: domain;

% If a vehicle belongs to the previous day, then its place in the schedule is the same as its place in the "vehicles" array.
% Both constraint-versions are kept to test which one is faster.
constraint forall (i in 1..numVehicles) ((vehicles[i,Date]=0 /\ schedule[i] != i) = false) :: domain;
%constraint forall (i in 1..numVehicles) (vehicles[i, Date] = 0 -> schedule[i] = i) :: domain;

% This constraint ensures there are no sequences of vehicles of the same paint color in the schedule
% that are longer than allowed by paintBatchLimit. However, as a preprocessing measure, the constraint
% is only applied if there is at least one colour, for which the number of vehicles to be painted with it
% is more than the paintBatchLimit.
% nvalue := returns the number of distinct values in the array
constraint
  if (forall (c in 1..numOfColours)(count(vehicles[..,PaintColour],c) <= paintBatchLimit))=false then
    forall (i in numPreviousDayVehicles+1..numVehicles-paintBatchLimit)
      (
         nvalue([vehicles[schedule[j] , PaintColour] | j in i..i+paintBatchLimit]) > 1
      )
  endif :: domain;

% For each contraint...
constraint forall (c in Constraints)
% Count the violations of the current constraint in the whole schedule. Start by setting the starting points
% of the windows you are about to examine, from the first to the last one. For example, if the current constraint ratio
% is N/P, the starting point of the first window is #previousDayVehicles - P + 2, and the starting point of the last window is
% #totalVehicles - N, as indicated in the problem definition.
  (violations[c] = sum(i in (numPreviousDayVehicles - ratios[c,Denominator] + 2)..(numVehicles-ratios[c,Numerator]))
      % For each window (positions i to i+P, unless we reach the end of the schedule), count the number of violations
      % of the current constraint, which is defined as "vehiclesWithConstraint - N". If the number of violations is negative,
      % then it is actually zero (0), so that is the usage of "max".
      (max(0,(
        sum (j in i..min(i+ratios[c,Denominator]-1,numVehicles)) (associatedConstraints[schedule[j],c])
             )-ratios[c,Numerator]
           )
      )
  % Domain consistency was too costly for this complex constraint, and so we decided bounds consistency was the way to go.
  ) :: bounds;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SYMMETRY BREAKING %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% The following function and constraint serve as our symmentry breaking constraint. The function
% "sameAttributes" returns true if two vehicles have the same contraints and paint colour.
% The constraint ensures that all these identical vehicles are ordered within our schedule in
% descending order (in terms of vehicle ID). The reasoning behind this is given in the report.
% The constraint is left commented-out because, for certain big files, it seems to work counter
% productively. More on this in the report.

function var bool: sameAttributes (var int: v1, var int: v2) = (vehicles[v1,PaintColour] = vehicles[v2,PaintColour] /\ forall (c in Constraints)
(associatedConstraints[v1,c] = associatedConstraints[v2,c]));

constraint symmetry_breaking_constraint(forall (i,j in numPreviousDayVehicles+1..numVehicles where i<j) (sameAttributes(schedule[i],schedule[j]) -> schedule[i] > schedule[j]));
```

```
144
145 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% OPTIMIZATION FUNCTION %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
146
147 % Our optimization function tries to minimize the sum of the 3 decision variables: high-priority constraints, low-priority constraints, and paintChanges.
148 % Each decision variable is multiplied by its corresponding multiplier, indicating its importance and priority.
149 % We attempted to use the below search strategy, for reasons explained in the report, but the results were worse than without the use of it.
150 % :: int_search(schedule, input_order, indomain_max, complete)
151 solve minimize (highViolations * objectiveMultipliers[HPRatioConstraints]) + (lowViolations * objectiveMultipliers[LPRatioConstraints]) + (paintChanges * objectiveMultipliers[PaintColourBatches]);
152
153 output[show(schedule),
154 "\n violations: ", show(violations),
155 "\n highViolations:", show(highViolations),
156 "\n lowViolations:", show(lowViolations),
157 "\n totalViolations:", show(sum(violations)),
158 "\n paintChanges:", show(paintChanges),
159 "\n minimizedGoal:", show((highViolations * objectiveMultipliers[HPRatioConstraints]) + (lowViolations * objectiveMultipliers[LPRatioConstraints]) + (paintChanges * objectiveMultipliers[PaintColourBatches])),
160 "\n", show((highViolations * objectiveMultipliers[HPRatioConstraints]) + (lowViolations * objectiveMultipliers[LPRatioConstraints]) + (paintChanges * objectiveMultipliers[PaintColourBatches]))];
```

B.3 Description

Redundant Decision Variables and Channelling Constraints. We do not think that redundant decision variables could be usefully introduced into our model, which thus has no channelling constraints. We didn't see of any way the compiler and backend would benefit from an extra variable, since the current formulation made all the information easily accessible and without the need to travel through a sequence data structures to reach it.

Implied Constraints. We did not introduce any implied constraints in our model, mainly because almost every rule was simply defined as a direct equality constraint: The number of violations per constraint was equal to the counted violations in our constraint at line 114. Then, the number of high/low-priority violations was equal to the amount of high/low-priority violations in the total counted violations, enforced through another equality constraint at lines 65 and 71. The same rule applied for the number of paint changes. This formulation was easy for the reader to understand, and we didn't think of a more implicit formulation that would benefit the compiler and backend itself.

Symmetry-Breaking Constraints. If certain vehicles are identical in terms of their attributes (high/low-priority constraints, day and colour) then any permutation of these vehicles within the day's schedule will provide the same outcome. Thus, the following constraint strictly requests that any such set of vehicles is scheduled in descending order, from the vehicle with the highest ID to the vehicle with the lowest one, eliminating, in this way, all other possible permutations of this set:

```
function var bool: sameAttributes (var int: v1, var int: v2) =
  (vehicles[v1,PaintColour] = vehicles[v2,PaintColour] /\ forall (c
  in Constraints)
(associatedConstraints[v1,c] = associatedConstraints[v2,c]));

constraint symmetry_breaking_constraint(forall (i,j in
  numPreviousDayVehicles+1..numVehicles where i<j)
  (sameAttributes(schedule[i],schedule[j]) -> schedule[i] >
  schedule[j]));
```

We used a function to assist us in ensuring two vehicles are indeed identical in terms of their attributes. The reason we chose descending order over ascending is explained in paragraph "Search Annotation".

As recommended, the constraint is flagged using the `symmetry_breaking_constraint` special predicate.

Variable-Constraint coupling Annotations. We had a few decision variables whose values would be completely determined by other parameters or variables, namely "highViolations", "lowViolations" and "paintChanges". To these variables, we appended the annotation "is_defined_var", and also appended the annotation "defines_var()" to their corresponding constraint, in order to help the backend.

Inference Annotations. Some constraints are used to provide a value to one of the decision variables in the objective function. Other constrains enforce the rules of the problem. For the latter constraints, we decided (through both testing and intuition) that explicitly requesting domain consistency would be to our interest: for example, the all_different(schedule) constraint should really make a difference when set to domain consistency, as it rules out an option for every other cell in the "schedule" array every time a cell is assigned a value. Even though that difference was not clearly felt in terms of speed of execution, it also didn't have any negative impact on the runtime, and so we kept it.

The only constraint to which we explicitly assigned bounds consistency was the constraint at line 114, counting the violations and implicitly deciding on the values of the "schedule" array. Through testing, we realised that domain consistency for this constraint was extremely harmful to our runtime, which was already expected since the constraint is quite complex and applying such a strict consistency is not simple.

Search Annotation. The following was the intended search strategy to be used:

int_search(schedule, input_order, indomain_max, complete).

The main decision variable to be determined, that would in turn give values to all the other decision variables in the objective function, was the "schedule" array.

The variable selection strategy was input_order since our schedule contained the previous day vehicles as well, and it seemed wise to start from the previous day cells/variables (which were already set to constant values through another constraint) and move our way up to the end of the current day.

The value selection strategy was indomain_max. The reason for this is that we noticed the default behavior of the compiler was to place the high-ID vehicles in the early cells of the "‘schedule” array; we didn't know why but it worked well enough, and so we decided to keep this setting. Additionally, this is the behavior that caused us to set our previously mentioned symmetry-breaking constraint to be of descending order: it would be unwise to start the value assignment from the top of the domain, while at the same time requesting an ascending order of values in the cells.

Unfortunately, the above search strategy only made the runtime worse. We tried many other combinations of settings and nothing worked as promisingly as the default configuration of the backend. Thus, we abandoned our attempts to manually and explicitly set the search strategy.

Pre-processing. We have noticed some instances with an extremely high paint batchLimit, to the point where there are not even enough vehicles with the same color to reach this limit. Thus, we wrapped our batchLimit hard constraint (which ensures the batchLimit is not exceeded by repeated same-color vehicles in the schedule) with an ”if” statement that only applies the constraint if there is at least one colour such that the total number vehicles with that color are more than the batchLimit:

```
constraint
  if (forall (c in
    1..numOfColours)(count(vehicles[..,PaintColour],c) <=
    paintBatchLimit))=false then
     forall (i in
       numPreviousDayVehicles+1..numVehicles-paintBatchLimit)
        (
           nvalue([vehicles[schedule[j] , PaintColour] | j in
             i..i+paintBatchLimit]) > 1
        )
  endif :: domain;
```

B.4 Implementation

The described model, with the prescribed comments, is uploaded as file CSP_solution.mzn.

Compilation and Running Instructions. To compile and run CSP_solution.mzn one must supply a data file in the command line. The model can be compiled and run by typing minizinc CSP_solution.mzn DATAFILE.dzn at the command line. --solver can be used to test different solvers, by providing the solver id.

Sample Test-Run Command.

```
> minizinc CSP_solution.mzn test.dzn
[1, 2, 3, 4, 5, 16, 15, 20, 19, 14, 13, 18, 17, 12, 11, 10, 9, 8, 7, 6]
 violations: [14, 4, 4]
 highViolations:18
 lowViolations:4
 totalViolations:22
 paintChanges:5
 minimizedGoal:180504
180504
----------
```

C Evaluation

We have chosen the backends for Gecode, Chuffed and MIP. The other solvers did not work in the same straightforward way the above solvers did.

Experiments. Table 1 gives the results for different data files (.dzn) on our RoadDef'05 model. For each file, we look at the results when the symmetry-breaking constraint was included (w/), and when it was not (w/o). The time-out was 10 minutes.

Apart from the test data given to us, we also created our own test file as a representative for small instances. The file is the following:

objectiveMultipliers = [10000,1,100];
paintBatchLimit = 8;
Constraints = {HPRC1,HPRC2,LPRC1};
ratios = $[||1, 5, 1||2, 4, 1||1, 3, 0||]$;
associatedConstraints =
$[||1, 0, 0||1, 0, 0||1, 0, 0||1, 0, 0||1, 0, 0||1, 0, 0||1, 0, 0||1, 0, 0||1, 0, 0$
$||0, 1, 0||0, 1, 0||0, 1, 0||0, 1, 0||0, 1, 0||0, 1, 0||0, 1, 0||0, 1, 0$
$||0, 0, 1||0, 0, 1||0, 0, 1||0, 0, 1||]$;
vehicles = $[||0, 1||0, 1||0, 1||0, 1||0, 1$
$||1, 1||1, 1||1, 1||1, 2||1, 2||1, 2||1, 2||1, 2||1, 2||1, 2||1, 2$
$||1, 3||1, 3||1, 3||1, 3||]$;
numVehicles = 20;

Technology	CP		LCG		MIP	
Solver	Gecode		Chuffed		COIN-BC	
Backend	mzn-gecode		mzn-chuffed			
`file (.dzn)`	`solution`	time(s)	`solution`	time(s)	`solution`	time(s)
test w/o symmentry	120704	t/o	80902	1.633	80902	4.43
test w/ symmentry	80902	19 - t/o	80902	0.592	80902	137
028_ch2_EP_RAF_ENP_S23_J3 w/o symm (92 vehicles - B Data)	781	43 - t/o	181089	0.20 - t/o	571	430 - t/o
028_ch2_EP_RAF_ENP_S23_J3 w/ symm (92 vehicles - B Data)	781	123 - t/o	–	t/o	–	t/o
035_ch1_RAF_EP_ENP_S22_J3 w/o symm (133 vehicles - B Data)	75994	1.3 - t/o	–	t/o	–	t/o
022_3_4_EP_RAF_ENP w/o symm. (499 vehicles - A Data)	6902	14 - t/o	–	t/o	–	t/o
022_3_4_EP_RAF_ENP w/ symm. (499 vehicles - A Data)	–	t/o	–	t/o	–	t/o
024_EP_RAF_ENP_S49_J2 w/o symm (1338 vehicles - X Data)	712527	386 - t/o	–	t/o	–	t/o
024_EP_RAF_ENP_S49_J2 w/ symm (1338 vehicles - X Data)	–	t/o	–	t/o	–	t/o

Table 1: Results for our RoadDef'05 model. In the 'time' column, if the reported time is less than the time-out (10 minutes here), then the reported objective value in the '`solution`' column was *proven* optimal; else the time-out is indicated by 't/o' and the reported objective value is either the best value found, but *not* proven optimal, before timing out, or '–', indicating that no feasible solution was found before timing out. If there is a number to the left of 't/o', that represents the time it took for the best value to be found, even though a timeout occured.

Analysis. Let us first summarize our conclusions, not only from the above data, but also from the many experiments that have been run on this model:

- The Gecode solver is the most reliable one when it comes to finding solutions, especially without our symmetry-breaking constraint. Yes, it is performing the worst on small instances, but on large instances (100-1500 vehicles) it always finds some solution before timing out, and it finds its first solution quite fast, too (it only took 14 seconds for the solver to find the solution 6902). Additionally, the found solutions are usually quite good, meaning the optimal has not / should not have a large difference from the best one found. We do not know why introducing our symmetry-breaking constraint on large instances harms the solver that much, but more on this later.
- The Chuffed and COIN-BC solvers perform quite well on small instances, much better than the Gecode one . Additionally, for small instances, our symmetry-breaking constraint harms the COIN-BC solver, benefits the Chuffed solver, and significantly benefits the CP solver. We can observe the latter by the fact that, for the test.dzn file, the CP solver + symmetry found the best solution (without terminating) in 19 seconds, while without the symmetry, it found a quite worse solution close to its 10 minute timeout.

Redundant Decision Variables and Channelling Constraints. Our model was argued in Section B.3 not to benefit from any redundant decision variables and their channelling constraints, hence there is no runtime impact analysis to make.

Implied Constraints. Our model was argued in Section B.3 not to benefit from any implied constraints, hence there is no runtime impact analysis to make.

Symmetry-Breaking Constraints. In the code handed-in, we left our symmetry-breaking constraint commented-out, since, for big instances, it greatly increased both the compilation time and the time it took for the solver to display its first solutions. Initially, that made us worry that our constraint was wrong, but then we tested it on smaller instances, such as our custom test file, and we witnessed either a smaller runtime, or solutions being found a lot faster, as can be seen from the table above. Our test file is made up of 3 sets of identical vehicles, which provided a really good test case for our symmetry, and indeed, such instances are significantly benefited by the inclusion of the constraint. While the Gecode and Chuffed solver seemed to really benefit from the constraint for small instances, the COIN-BC solver did not, most probably because of its nature (its an MIP technology, which benefits more from relaxations rather than other aspects of a formulation).

Inference Annotations. The impact of our inference annotation is not experimentally demonstrated here, since the effects of the annotations were only visible by calculating the time it took for the first few results to appear. For example, for most constraints, changing the consistency from bounds/default to domain consistency didn't negatively impact our performance, and so we chose the latter. For the violations-counting constraint, however, at line 114, domain consistency hindered us from seeing any results appear for a long time, in contrast to bounds consistency, and thus we decided on the bounds one.

Search Annotation. The impact of the search strategy we suggested
int_search(schedule, input_order, indomain_max, complete)

is not presented here, since the effects of the addition of the annotation were only visible by calculating the time it took for the first few results to appear. For example, when testing out the data file "022_3_4_EP_RAF_ENP.dzn" without the strategy, the first 5 results appeared within a few seconds, but when we tested with the strategy, no results appeared on the screen for a long time. Neither of the attempts finished within a reasonable timeout to at least compare their finishing times.

General suggestions to the manager. In short, we would suggest this model because of its maintanability/extensibility; the code and model formulation are quite user-friendly. As for the solvers and technologies to go with it, we would suggest the CP solver as the main reliable solver for all the sizes of instances. In case of small instances (about 50 vehicles), the Chuffed and COIN-BC solvers could perform better, and if, in addition, a lot of vehicles are identical, the introduction of the symmetry constraint could definitely help.

Feedback to the Teaching Staff

The amount of time we had to get comfortable with MiniZinc (2 weeks) felt not enough to be able to solve problems of this scale optimally. We had a lot of ideas but expressing them in MiniZinc proved difficult, and the documentation didn't really help as much as it could: to find the functions we were looking for, we had to use keywords in a search field, but it was often not easy finding the correct keywords that expressed that. Maybe the documentation in itself wasn't badly structured, but the lack of forums and sites discussing MiniZinc matters left us with only the documentation as our source of knowledge about the tool (and the slides, which were pretty helpful, no doubt).

The assignment was really interesting in itself.

Experimentation was tedious because of the long waiting times: the only way to realize whether a change was beneficial or not was to calculate different runtimes in terms of minutes, so we needed, for example, an hour to figure out whether a change had reduced the runtime significantly.