Automated Reasoning Report

Enrico Santi

January 2023

Contents

1 Introduction to the problem

The COP we are given concerns the admissible seat assignments we can make with n prisoners given k tables where at most 8 people can sit. We are asked to schedule an assignment for each day of the week, with some requirements we must meet, such as the fact that we can't let prisoners considered dangerous in the same table the day after or that for safety reasons if a prisoner is dangerous a policeman must also be present at the table. The objective is to minimize the number of policemen we need to be sure all the safety conditions are met.
For completeness the full description of the problem is reported below:

Jailcanteen

There are n jail prisoners $p_1, \ldots, p_n$ and it is known a set of facts of the form $\texttt{hate}(p_i, p_j)$ and a further list of facts $\texttt{dangerous}(p_i)$, for some $i, j \in 1..n$.

A canteen table can contain at most 8 people. There are k tables (input data). In your instances choose $k = \frac{n}{6} + 1, \frac{n}{6} + 2, \frac{n}{6} + 3$.

You can put at most one pairs of prisoners that hate each other in the same table. Tables without dangerous prisoners do not require a policeman. In the other cases a policeman is needed in that table (thus the table capacity is reduced to 7).

The problem is to finding a set of assignment for prisoners in tables for all the week. Any pair of mutual haters can be sit together at most once per week. Dangerous prisoners have to change table every day (and can sit in a previusly used table after at least three days).

Look for a solution with the minimum number of policemen.

Figure 1: The original script of the problem

1.1 The constrains

To ease the modelling phases (both constraint programming and ASP) we report now briefly all the constraints the seat assignments must satisfy:

- At each table at most 8 prisoners can sit.
- Each table can have none or one pair of prisoners that hate one the other.
- If a table presents a pair prisoners (e.g. p_i and p_j) hating each other, then p_i and p_j must not be sitting together any other day of the week.
- If a table has one or more dangerous prisoners a policeman must be also present.
- Dangerous prisoners must change table every day and they can sit to a previously used table not before three days.

It's easy to find a trivial upper bound to the number of policemen required (if an instance is satisfiable) and it's the number of tables (k). Furthermore in the original script there are specified the three values for k to use in the different instances.

1.2 Assumptions

We assume that regardless the number of dangerous prisoners at a table, at most a single policeman is sufficient to be present (even if at a table we have 7 dangerous prisoners). We assume the relationship *hate* to be symmetric, i.e. if p_i hates p_j also $p_i$hates p_j, we also assume that nobody hates himself. Lastly we assume, with reference to the last constraint, that already on the third day the dangerous prisoner can reuse the previously used table (e.g. only two 'full days' must pass, so if p_i uses the table t_j on Monday, already on Thursday he can sit again at t_j).

2 Modelling with constraint programming

Let now consider the model defined in MiniZinc to describe the problem and the reasons for the design choices made. The code is presented below.

Listing 1: The model defined

```
%define the instance:

%we set the number of prisoners
par int: n;
%set the number of tables, we use the ceiling function
par int: k=ceil(n/6)+3;

%we describe whether a prisoner is dangerous or not by encoding it in a binary array.
array[1..n] of par 0..1:isDangerous;

%we define the hate predicate with a symmetric matrix n*n
array[1..n,1..n] of par 0..1:hates;

%to keep track of who sits where in which day
array[1..k,1..n,1..7] of var 0..1:seats;

%we keep track of the officers needed via the following matrix
array[1..k,1..7] of var 0..1: officers;

%--------------------------------------------------------------------------------%
%constraints:


%each prisoner must be in one and only one table (for each day)
constraint forall(prisoner in 1..n, days in 1..7) (sum(table in
    1..k)(seats[table,prisoner,days])=1);
```

```
%each table has at most 8 seats, 7 if someone is dangerous (next constraint, they could be
    unified with an if, but keeping them split it's easier to read)
constraint forall(table in 1..k,days in 1..7) (sum(prisoner in
    1..n)(seats[table,prisoner,days])<=8);

%if there's at least one dangerous prisoner in a table we use one seat for an officer, and
    we keep track of it
constraint forall(table in 1..k,prisoner in 1..n, days in 1..7) ((isDangerous[prisoner]=1
    /\ seats[table,prisoner,days]=1) -> ((sum(prisoner2 in
    1..n)(seats[table,prisoner2,days])<=7) /\ officers[table,days]=1));

%each table has at most one pair of haters (if a couple hates each other and they are sit
    at the same table) it can't be that there exist a third hating one of the two, or two
    different other people hating each other
%we can use indexes like: prisoner2 in prisoner1+1..n since the relationship is symmetric,
    we don't need also to check whether p4!=p3 and so on since nobody hates himself
    (hopefully)
constraint forall(prisoner1 in 1..n,prisoner2 in prisoner1+1..n,table in 1..k, days in
    1..7)
              ((hates[prisoner1,prisoner2]=1 /\ seats[table,prisoner1,days]=1 /\
                  seats[table,prisoner2,days]=1)->
              forall(prisoner3 in prisoner1+2..n,prisoner4 in prisoner2+1..n)(
                    (seats[table,prisoner3,days]=1 -> hates[prisoner1,prisoner3]=0 /\
                        hates[prisoner2,prisoner3]=0)
                    /\
                    (seats[table,prisoner4,days]=1 -> hates[prisoner1,prisoner4]=0 /\
                        hates[prisoner2,prisoner4]=0
                         /\ (seats[table,prisoner3,days]=1 ->
                             hates[prisoner3,prisoner4]=0))
                    ));

%haters can be together at most once a week (if p1 hates p2 and they sit together today
    for every other day they must sit away)
constraint forall(prisoner1 in 1..n,prisoner2 in prisoner1+1..n,table in 1..k, days in
    1..7)
              ((hates[prisoner1,prisoner2]=1 /\ seats[table,prisoner1,days]=1 /\
                  seats[table,prisoner2,days]=1)                    ->(forall(days2 in
                  days+1..7)(not(seats[table,prisoner1,days]=1 /\
                  seats[table,prisoner2,days]=1))));

%dangerous prisoners must change table
%if a dangerous prisoner sits here today, it must not sit here tomorrow or the day after
    tomorrow
constraint forall(prisoner in 1..n,table in 1..k, days in 1..6)((isDangerous[prisoner]=1/\
    seats[table,prisoner,days]=1)->((seats[table,prisoner,days+1]=0 /\
    seats[table,prisoner,days+2]=0) \/ days+1>=8));

solve minimize (max([sum(i in 1..k) (officers[i,j]) | j in 1..7]));
%-----------------------------------------------------------------------------------------%
```

```
%create array just for output purposes
array[1..7] of par string: day=["Monday ","Tuesday ","Wednesday ","Thursday ","Friday
    ","Saturday ","Sunday  "];

%output: for each day we show which prisoner is in which table (by putting a one in the
    table i under the cell j to state that prisoner j is in the table i)

%the ' else "" ' are just put to suppress warnings
output
    [if (tables=1 /\ prisoner=1) then format(day[days]) else "" endif ++
     if ( prisoner=1) then " table_"++format(tables)++ ": " else "" endif ++
     if (fix(seats[tables,prisoner,days])=1) then format(prisoner)++" " else "" endif ++
     if (tables=k /\ prisoner=n) then "\n" else "" endif | days in 1..7, tables in 1..k,
         prisoner in 1..n];
```

As it can be seen the code is divided into three macro regions, the first which allows to specify the instance to solve, the second specifying the constraints and the objective function and a third one for the output part. The instance is described by the number **n** of prisoners, the number **k** of tables and four arrays:

- **isDangerous**: the array describing which prisoner is dangerous, p_i is dangerous iff isDangerous[i]=1.
- **hates**: the symmetric matrix describing whether the prisoner i hates j and vice versa (hates[i,j]=1).
- **seats**: the matrix containing for each table, for each day, whether the prisoner i is (1) or not (0) present at table j.
- **officers**: the matrix that describes whether an policeman is needed at table j on day i.

For what concerns the constraints used, the explanation is straight forward. Each constraint presented in the subparagraph 1.1 is mapped with one (more or less complex) constraint in MiniZinc. In addition one more constraint (which in 1.1 wasn't mentioned since it's common sense) is the fact that each prisoner must be in one and only one table each day.
One detail to notice is that in the last constraint, in the consequent of the implication there is a disjunction, in which the second term is $days + 1 \geq 8$. This is added because the loop exceeds the bounds of the array **seats**, since we access the index days+2 (which with days ranging up to 6, is equal to 8). MiniZinc indeed tells us in a warning that:

> Undefined result becomes false in Boolean context.

For this reason, when days is 6 the implication may fail since the second part of the conjunction is by default false. This second part is set to true when days is 6 by the disjunction added to make sure there are no false negatives. An alternative way to deal with this issue would be to make *days* in the forall range up to 5, and add another constraint to deal with the edge case *day=6*. This alternative has been avoided to be able to map the constraints 1.1 to one MiniZinc constraint. A more exhaustive description for each constraint is written as a comment in the source file.
The output presents, for each day, for each table the prisoners at that table. Notice that no global constraints were used, the model could be modified to include them hoping an improvement in

performances.
The code above doesn't present an initialization for the parameters, which could be easily done via another file (generated by a Python script) to save time.

2.1 Alternative models

There are many other ways to write a model for the problem, which can lead to more or less efficient searching for solutions. One possibility could use instead of the matrix **seats** a matrix **tableSeats** ranging in **[1..8,1..k,1..7]** and containing values from -1 to n. In this way we could describe for each of the eight seats of each table who occupies it, with 0 being an empty seat, -1 being a policeman and i ranging from *1 to n* being the *i-th* prisoner. There would be no need for the **officers** array, since the information could be obtained from **tableSeats**. The constraints and the objective function should be modified accordingly.
The main reason for choosing the model presented and not the alternative model is because throughout the course of Operational Research emphasis was put on the usage of numbers and labels represented by numbers and to avoid the second case when possible.

2.2 Some results

In this section some results achieved will be presented. All the tests are done on the same computer trying to recreate the most uniform environment possible (i.e. no other user tasks in background). The computer presents an Intel I3 4005u with two physical cores (presenting Hyperthreading, so capable of dealing with 2 threads with each core). Let's see some results distinguishing the instances into three macro categories according to their dimensions:

- Small ($n \in (2, 10]$): In this case both GCode (6.3.0) and Chuffed (0.10.4) are capable of solving the instances in a couple of seconds (compilation included). Having tested some instances though (both satisfiable and not) Chuffed seems to be faster by two or three seconds (where GCode takes in General 5 or 6 seconds). To match the performances of Chuffed, GCode can be specified to use more threads, by using two of them the performances are comparable to those of Chuffed. The instances tested were the first six of the small folder, with some small variation to make some of them satisfiable.

- Medium ($n \in (10, 20]$): In this case the average run-time for some of the instances tested with the same configuration as before is 15 seconds for Chuffed and 18 for GCode.

- Big ($n \in (20, 30]$): Again both solvers are able to solve the problem instances. GCode on three threads performs on average 20 seconds worst than Chuffed, but both for the majority of the instances tested remain below three minutes of run-time.

- HUGE ($n \in (30, 40]$): Most of the instances in this range, especially when n is around 40 can't be solved within the 5 (or even 6) minute time limit. Also trying different configurations for the solvers or increasing the number of threads used by GCode to four doesn't lead to many of these instances being solved within the limit (on the machine specified). Some partial solutions presented the optimum found so far to be quite high, such as 6 or 7 officers, which may tell us that the search was only in the first stages (or we just weren't lucky).

Regardless of the instance dimension, the number of tables addressed by the exercise (i.e. in the order or $\frac{n}{6}$) in most of the cases tested seems to be too small to admit a solution. It could be that

on non random instances (more realistic) with less people hating each other this number of tables could be sufficient in most cases.

2.2.1 How to replicate the results

To replicate the tests above a Python script named 'minizincInstances.py' is provided, when executed it will generate the same 30 instances used (having fixed the seed) for each size. It's then possible to invoke MiniZinc and specify one of the generated files as input.

3 Modelling with ASP

Let now describe the model defined to describe the problem, the code is presented below.

Listing 2: The model defined

```
day(1..7).
table(1..k).
prisoner(1..n).

%INPUT: define who is dangerous and who hates each other

isDangerous(1).
isDangerous(2).

hates(1,2).
hates(2,1).

%as in MiniZinc we must state that for each day, each prisoner must be exactly in one table
1 {atTable(T,P,D) : table(T)} 1 :- prisoner(P),day(D).

%maximum 8 people per table
0{atTable(T,P,D) : prisoner(P)}8 :- table(T),day(D).

%only seven if someone is dangerous, 1 is then an officer
0{atTable(T,P1,D) : prisoner(P1)}7 :-
    table(T),day(D),prisoner(P),isDangerous(P),atTable(T,P,D).

%if someone is dangerous there's an officer at that table
officer(T,D) :- table(T),day(D),prisoner(P),isDangerous(P),atTable(T,P,D).

%each table has at most one pair of haters (we remove now the case in which p1 (or p2, by
    symmetri) hates two at his table)
:-prisoner(P1),prisoner(P2),hates(P1,P2),day(D),table(T),atTable(T,P1,D),atTable(T,P2,D),prisoner(P3),
P3!=P2, atTable(T,P3,D), hates(P1,P3).
%it's not necessary to write P2!=P1 since no one hates himself (for this reason also
    P3!=P1 isn't necessary)

%each table has at most one pair of haters (we remove now the case in which p1 hates p2,
    but in the same table we have other two people hating each other)
:-prisoner(P1),prisoner(P2),hates(P1,P2),day(D),table(T),atTable(T,P1,D),atTable(T,P2,D),prisoner(P3),
```

```
prisoner(P4), P3!=P2, P3!=P1,P1!=P4, P2!=P4, atTable(T,P3,D), hates(P1,P3).
%as above we don't need to compare P1 and P2 or P3 and P4

%haters can be together at most once a week, which means there can't be two different days
    in which P1 and P2 who hate each other meet at one table (which doesn't need to be
    always the same)
:-prisoner(P1),prisoner(P2),hates(P1,P2),day(D1),day(D2),table(T1),atTable(T1,P1,D1),
atTable(1,P2,D),atTable(T2,P1,D2),atTable(T2,P2,D2), D1!=D2 ,table(T2).

%dangerous prisoners must change table and can't reuse it before three days
:-table(T),day(D),prisoner(P),isDangerous(P),atTable(T,P,D),atTable(T,P,D+1),atTable(T,P,D+2).

%minimizziamo
counterOfficers(N) :- N = #max{T: officer(T,_)}.
#minimize {N:counterOfficers(N)}.
```

As it can be seen the code is much shorter and more readable than the one written in MiniZinc, even though ASP wasn't born for constraint programming. As for the previous model, the code could be subdivided into three main sections. The first one is related to the **initial state** (i.e. the knowledge we input), then we have the different predicates defining the constraints and at the end we recreate predicate defining the function to minimize.
The first rule is equivalent to the first constraint of the MiniZinc model, which specify that every prisoner in a given day must sit somewhere (and just in one place). The second and third rule specify what was imposed by the second and third constraint of the MiniZinc model, i.e. there can be at most eight prisoners per table, or seven if there's someone dangerous. The third rule completes what was specified in the third constraint in the other model, so the fact that if a prisoner is dangerous, an officer is needed at that table. The remaining rules are negations which respectively make impossible that at one table there are more than a couple of prisoners hating each other (fourth and fifth rule), make impossible for two hating prisoners that already met at a table to meet again (sixth rule) and make impossible to not move dangerous prisoners.

3.1 Some results with ASP

As for the model above instances are generated randomly through a script. This time though, not being able to give Clingo a file where the initial knowledge (**initial state**) is specified, many instances of the same problem were created by modifying the first part of the .lp file to specify different situations. Both the instances and the script ('createInstances.py') that generated them are provided. To test the instances is sufficient to call Clingo on them.
The results achieved with ASP seem to be much better than the ones achieved with MiniZinc, indeed both with small and large instances (e.g. 'instance_29.pl') the search time remains under 10 seconds. In particular for instances up to n=20 the time required is in the order of hundreds of milliseconds, while when considering instances from 20 to 30 grows in the order of seconds.

A possible explanation for this phenomenon is briefly presented in the solutions.

4 Conclusions

To sum up both the methods allow to solve many instances of the problem presented. When the instances start to grow over the size n=35 the time required starts to increase a lot for the MiniZinc model. One possible continuation could be to change the model in MiniZinc and use the alternative one presented alongside with some global constraints to see if the performance increase on larger instances. Another possibility would be to try the MiniZinc model on a different machine with more cores to exploit the great parallelizability of the search phase.

4.1 Why ASP seems faster than MiniZinc

The model written in ASP seems to be more readable and compact even though ASP started out with a different objective (it even seems a lot faster). An hypothesis that could explain why this second model is faster could reside in the instances not the model itself, indeed the instances generated via 'createInstances.py' seem to to be simpler, in the sense that they present less dangerous prisoners or haters (compared to instances generated with the other script given same the number n of prisoners).
Another possible explanation could just simply be that the model presented for MiniZinc is not efficient (and a much better one could be written) compared to the one in ASP.