Automated Reasoning Assignment

mattia guiotto

147654@spes.uniud.it

Contents

1 Introduction

The goal of this assignment is, given a starting problem, to model it with both logic programming, more precisely using the *answer set programming (*ASP*)*, and as a constraint optimization problem, using the constraint modeling language *Minizinc*.

1.1 the starting problem

The problem we are asked to solve is the following:

The tavern keeper has k rooms with 4 tables each. Each table has 4 seats. Among his guests that have booked a table $p_1, \ldots, p_n$ he has some knowledge of the following kind:

- A set of predicates same_table(p_i, p_j) representing that p_i and p_j belong to the same group (they are already clusters of at most 4 people, they know in advance that tables have four seats),
- A set of predicates hate(p_i, p_j) representing that p_i and p_j hate each other,
- A set of predicates dangerous(p_i), representing that p_i is a dangerous person,

for some $i, j \in \{1, \ldots, n\}$ and $x \in \{1, \ldots, k\}$. Among the k rooms there are some covered by a security cameras.

We have to find an assignment for customers in tables in order to minimize the number of rooms and fulfill the following constraints:

[SAME-TB] People of the same group have to sit in the same table,

[HATE] avoid putting people who hate each other in the same room,

[DANG] dangerous people must be located in rooms with security cameras.

Moreover, if a table is occupied by a group of one or two people, a further (not dangerous) group of one or two people can be sit in the same table.

2 ASP modeling

The input to our model comprises the amalgamation of sets of facts characterized as follows:

1. $\text{room}(1..k)$.
2. $\text{table}(1..4 * k)$.
3. $\text{person}(1..n)$.
4. $\{\text{same\_table}(i, j).: i, j \in \{1, \ldots n\}\}$
5. $\{\text{hate}(i, j).: i, j \in \{1, \ldots n\}\}$
6. $\{\text{security\_cameras}(x).: x \in \{1, \ldots k\}\}$
7. $\{\text{dangerous}(z).: z \in \{1, \ldots n\}\}$

with $n, k \in \mathbb{N}$. We let $\text{room}(1..k) = \{\text{room}(1), \ldots, \text{room}(k)\}$, $\text{table}(1..4 * k) = \{\text{table}(1), \ldots, \text{table}(4*k)\}$, and $\text{person}(1..n) = \{\text{person}(1), \ldots, \text{person}(n)\}$. Their names easily suggest their respective representations. Each instance of the model differs on n, k, as well as the union of sets of this nature. Now, transitioning to predicates found in the head of rules which represent constraints outlined in the problem statement, the following *cardinality constraints* rules were employed:

- $1\{\text{is\_in}(T, R) : \text{room}(R)\}1 : -\text{table}(T).$
- $4\{\text{is\_in}(T, R) : \text{table}(T)\}4 : -\text{room}(R).$
- $1\{\text{assign}(X, T, R) : \text{table}(T), \text{room}(R), \text{is\_in}(T, R)\}1 : -\text{person}(X).$
- $\text{count\_seats\_occupied}(R, T, S) : -\text{table}(T), \text{room}(R), \text{is\_in}(T, R)$
  $, S = \#\text{count}\{X : \text{person}(X), \text{assign}(X, T, R)\}.$
  $\text{bad\_assignment} : -\text{table}(T), \text{room}(R), \text{is\_in}(T, R),$
  $\text{count\_seats\_occupied}(R, T, S), S > 4.$
  $: -\text{bad\_assignment}.$

They effectively capture the given scenario where a table T belongs to exactly one room R, each room has exactly 4 tables, each person is assigned to exactly one table in a room, and no more than 4 people can be seated at any given table T. It's important to note that the aggregate function #count is utilized to determine the number of people seated at a table T, while the predicate count_seats_occupied(R,T,S) is employed to store the count of people seated at table T in room R. Furthermore, the predicates is_in(T, R) and assign(X, T, R) respectively signify that table T is located in room R and person X is seated at table T in room R. The constraints **[SM-TB]**, **[HATE]** and **[DANG]** are modelled as follows:

[SAME-TB]

$$\text{equal}(T1, T2, R1, R2) : -\text{table}(T1), \text{table}(T2), \text{room}(R1), \text{room}(R2),$$
$$\text{is\_in}(T1, R1), \text{is\_in}(T2, R2), T1 = T2, R1 = R2.$$

$$: -\text{person}(X), \text{person}(Y), X! = Y, \text{same\_table}(X, Y), \text{assign}(X, T1, R1),$$
$$\text{assign}(Y, T2, R2), \text{not equal}(T1, T2, R1, R2), \text{table}(T1), \text{table}(T2), \text{room}(R1),$$
$$\text{room}(R2), \text{is\_in}(T1, R1), \text{is\_in}(T2, R2).$$

[HATE]

$$: -\text{person}(X), \text{person}(Y), X! = Y, \text{hate}(X, Y), \text{assign}(X, \_, R), \text{assign}(Y, \_, R), \text{room}(R).$$

[DANG]

$$: -\text{person}(X), \text{dangerous}(X), \text{assign}(X, \_, R), \text{room}(R), \text{not security\_cameras}(R).$$

These constraints accurately depict the given scenario: ensuring that no two different people of the same group are seated at different tables, preventing individuals who harbor animosity toward each other from sharing the same table, and ensuring that dangerous individuals are not seated in rooms lacking security cameras. Lastly, the objective function, aimed at minimizing the number of rooms used,is encoded through the optimization statement #*minimize*:

$$\text{room\_used}(C) : -C = \#\text{count}\{K : \text{room}(K), \text{assign}(\_, \_, K)\}.$$
$$\#\text{minimize}\{C : \text{room\_used}(C)\}.$$

Note that the input instances for the model were generated by a C program. It takes care of ensuring that instances are not created where groups of people who have to be seated at the same table exceed four people. Additionally, it addresses the issue of not generating instances where a pair of people belonging to the same group harbor mutual animosity, as well as avoiding the generation of symmetric pairs. In cases where individual X bears animosity toward individual Y, only the pair hate(X, Y) is generated. Similarly, for instances where individual X belongs to the same group as Y, analogous considerations are made.

3 Minizinc modeling

The developed model is founded upon the same principles as the ASP model described earlier. It takes as input statements that specify the following parameters:

$$\begin{aligned}
&\text{int} : k = x;\\
&\text{int} : t = 4 * x;\\
&\text{int} : n = y;\\
&\text{array}[1..n, 1..n] \text{ of } 0..1 : \text{same\_table};\\
&\text{array}[1..n, 1..n] \text{ of } 0..1 : \text{hate};\\
&\text{array}[1..n] \text{ of } 0..1 : \text{dangerous};\\
&\text{array}[1..k] \text{ of } 0..1 : \text{security\_cameras};
\end{aligned}$$

for certain $x, y \in \mathbb{N}$. The parameters k, t, n respectively denote the count of rooms, tables and people. Within matrix "same_table", if same_table$[i, j] = 1$, it signifies that the person i and the person j have to sit at the same table. The matrix "hate" follows a similar interpretation. Lastly, the vectors "dangerous" and "security_cameras" depict the scenario where the person i is deemed dangerous if and only if dangerous$[i] = 1$, and there are security_cameras in the room y if and only if security_cameras$[y] = 1$, for some $y \in 1 \ldots k$ and $i \in 1 \ldots n$.

The decision variables for the model are as follows:

$$\begin{aligned}
&\text{array}[1..t] \text{ of var } 1..k : \text{is\_in};\\
&\text{array}[1..k, 1..t] \text{ of var set of } 1..n : \text{assign};\\
&\text{var int} : s;
\end{aligned}$$

The vector "is_in" serves to record room in which a table is located. If is_in$[z] = c$, it indicates that the table z is situated in the room c, where $z \in 1 \ldots t$, $c \in 1 \ldots k$. The matrix "assign" illustrates the arrangement where a person p is seated at table z in room c (denoted by p in assign$[c, z]$). The variable s represents the objective we aim to minimize. It denotes the count of rooms utilized to accommodate people.

[GOAL]

constraint $s = \text{count}(i \text{ in } 1..k)(\text{exists } (j \text{ in } 1..t \text{ where is\_in}[j] = i)(\text{assign}[i, j]! = \{\}));$

constraint $s < k$;

4 Search Euristic

By default, MiniZinc does not specify how solutions should be searched for, leaving this task entirely to the underlying solver. However, there are situations where we may need to define the search process. This necessitates communicating a search strategy to the solver. It's important to note that not every solver is obligated to support all potential search strategies. MiniZinc employs a uniform method for providing additional information to the constraint solver through *annotations*. These annotations are added to the solve directive, following the keyword *solve*. In my scenario, the following search strategy was utilized:

```
solve :: seq_search([int_search(is_in,occurrence,indomain_random),
set_search(assign, occurrence, indomain_random)])
 :: restart_linear(n)
minimize s;
```

Both the int_search and set_search annotations prioritize variables with the highest number of associated constraints, assigning them random values. Restarting the search proves significantly more resilient in locating solutions and prevents stagnation in unproductive search areas. However, it's worth noting that restarting the search is ineffective if the underlying search strategy doesn't deviate from its initial approach.

Regarding simpler instances, this strategy yields noticeable time improvements compared to the default approach. However, for medium and difficult instances, no strategy succeeds in achieving the optimal value within the given timeout.

5 Experimental Results

The runtime performance of the two proposed encodings was compared using a collection of benchmarks generated pseudo-randomly by a C program. The evaluations were conducted utilizing OR Tools CP-SAT solver version 9.8.3296 and clingo version 5.4.1, with a predefined timeout of 5 minutes. Additionally, MiniZinc was evaluated using alternative solvers compatible with the platform, namely Gecode version 6.3.0 and Chuffed version 0.13.1. However, despite efforts, both solvers, whether equipped with search annotations or not, exhibited significantly inferior performance compared to OR Tools on simpler instances. Consequently, they were deemed unsuitable for use. Neither solver was able to attain the optimal solution within the prescribed timeout for any instance.

In order to categorize the instances by their complexity, certain parameters were established:

[n] Number of people that need to be seated.

[k] Number of rooms.

[N] Consider the C program responsible for generating instances according to the proposed encodings. It operates by pseudo-randomly creating $N/2$ pairs of individuals to be seated together, $N/3$ pairs of individuals who harbor animosity towards each other, $N/3$ individuals considered dangerous, and $k/3$ rooms under surveillance by security cameras. At each iteration, it generates a pair and examines whether, in the case of seating together, either a symmetric pair already exists or the group to which one member of the new pair belongs already comprises four individuals. Similarly, in instances of animosity, it verifies whether the pair shares a table; if so, it eliminates the pair and proceeds. With increasing N, the complexity of the instances grows as the number of constraints governing individuals escalates.

Table 1 shows the corresponding time values of the two proposed encodings across different generated instances. Here is a brief explanation of the table's semantics: The column "Best" reports the best value found by the solver within the timeout. For certain instances, the solvers may have found a value, but it might not be the optimal value. The value in the "Time (s)" column indicates that the solver has found the optimal value for that particular instance within the timeout. Note that some cells in the "Time (s)" column are empty. This indicates that the solver did not find the optimal solution for that instance within the 300 second timeout. Consequently, for those instances, the term "UNKN" (unknown) is reported in the "Optimum" column. Furthermore, if the solver did not find any value for certain instances, the corresponding cell in the "Best" column is left empty.

5.1 Considerations

As can be easily seen from Table 1, the ASP model is clearly better across all instances than the MiniZinc model. During development, I initially implemented the ASP model and then the MiniZinc model following the ASP approach. I believe that the poorer performance of MiniZinc is due to the data structure used to store the decision variables, and therefore, with a more sophisticated modeling approach, it can achieve better results. Moreover, it is noteworthy that I initially expected that fixing n and N and increasing k, i.e., the number of rooms and hence, the number of tables and chairs available, would make the model easier. However, contrary to my expectations, it became harder (see instances 15, 20, 24).

Table 1: performance of ASP vs MiniZinc

					ASP		**MiniZinc**	
Instances	N	n	k	Optimum	Best	Time (s)	Best	Time (s)
EASY instances								
1	5	10	4	2	2	0.035	2	2.419
2	5	12	4	2	2	0.046	2	3.252
3	5	14	4	2	2	0.055	2	2.072
4	5	16	4	2	2	0.047	2	3.234
5	5	18	4	2	2	0.043	2	7.194
6	5	20	4	2	2	0.054	2	12.256
7	5	25	4	2	2	0.075	2	18.817
8	5	30	4	2	2	0.115	2	
9	10	20	4	UNS	UNS	0.366	UNS	2.162
10	10	25	4	2	2	0.090	3	
11	10	30	4	2	2	0.126	3	
12	25	25	4	3	3	0.610	3	
13	25	28	4	2	2	0.160	2	10.012
14	25	30	4	2	2	0.196	2	11.701
MEDIUM instances								
15	25	35	8	3	3	98.882		
16	25	33	8	3	3	85.274		
17	30	33	8	UNS	UNS	71.429		
HARD instances								
18	10	35	4	UNKN	3		3	
19	25	35	4	UNKN	3			
20	25	35	12	UNKN	3			
21	25	50	20	UNKN	4		4	
22	25	50	40	UNKN				
23	10	35	12	UNKN	3		3	
24	25	35	20	UNKN	3			

[1] UNS stands for unsatisfiable
[2] UNKN stands for unknown