Automated Reasoning May 2021

Project Report

Alex Della Schiava Matricola: 134309

1 Outline

In this document I intend to describe my ASP and Minizinc solutions to the 7-segments displays problem.
Section 2 offers a brief analysis of the problem, describing the main assumptions on which the solutions are based on.
Section 3 provides a thorough illustration of the Minizinc model, justifying my reasoning behind the main choices and providing references to the code.
Section 4 does pretty much the same as Section 3 but with respect to my ASP solution.
In Section 5 the results of the tests on the two models are shown, together with an illustration of the conditions in which the testing was done and the conclusions I have drawn from it.
Finally Section 6 offers some final remarks on what this project has achieved and on the possible optimizations and ideas that could make both the problem and the solution more interesting.

2 Introduction

Without going through the whole assignment, I would first like to denote part of the terminology used in this report. The parameters u, d, c, l denote the number of sticks of the allowed shapes, which here I refer to as, for example, *u-shaped sticks*.
Diving into the 7-segments display problem, the initial issue I had to face consisted in how many 7-segments displays a $n * n$ grid can contain. This number is strictly related to the meaning of *"touching displays"*. As for this solution, two displays are *touching* if they shared at least one vertex of the grid. Figure 1 offers a borderline example of this definition. In this example the *8* in the top-left corner and the *1* in the bottom-right corner are *touching* since their displays are sharing the vertex in the center.

This assumption makes it much simpler to provide a model for the problem and, at the same time, offers more interesting solutions. In fact, it is easy to notice how without this assumption, the majority of the solutions would consist in a sequence of *1*s and *7*s, since they occupy a much smaller space than all the other digits.
Finally, to answer how many 7-segments displays can fit in a $n*n$ grid, it is simply needed to calculate $\lfloor \frac{n+1}{3} \rfloor * \lceil \frac{n}{2} \rceil$. In the Minizinc

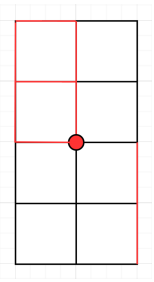


Figure 1: Two *touching displays.*

model, this number is referred to as `blocksNumber`.
On a last note: for both solutions pretty printing is supported for a better understanding of the solutions.

3 My Minizinc solution

3.1 The main data structure

Developing the Minizinc model, decided that the best idea was to start from a 2-dimensional array to represent the segments of each display in the grid and, more specifically, whether they are lit or not. This array (`isStick`) is defined at Line 10. Initially it was supposed to be an array of boolean variables, but then I thought it was for the best to use this same data structure to define by which type of stick (u, d, c or l) a segment was lit. In order to do this I decided to use an enumerated type `sticks`, defined at Line 8.

3.2 Shaping the digits

Lines 25 to 106 contain the predicates used to define the shape of the ten digits inside a display. To identify the segments of a display numbers from 1 to 7 are used in the way shown in Figure 2. There is no particular reason behind this choice, it simply seemed the more intuitive way of arranging the segments. The predicates defining the digits are all put into a disjunction in order to define the predicate `isNumber`, which is true when the segments of a display are lit in a way that corresponds to the shape of a digit and false otherwise. Notice that it is never the case that more than one digit's predicate is true for a given display, since the predicates of the digits are defined in a way that makes them exclude each other. One useful reason to have the `isNumber` predicate is that it allows to turn off all the segments whenever a display does not contain a digit. This prevents the solver from finding solutions with displays showing meaningless shapes whenever there are sticks leftover.

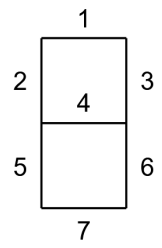


Figure 2: Numeration of the 7 segments.

3.3 Constraining u-shaped sticks

Constraining the u-shaped sticks was trivial, the only necessary constraint is that there may not be more than u segments lit by a u-shaped stick, that is, for all i, j there can be at most u cells in the array `isStick` such that `isStick[i,j]=isU`.
The implementation of this constraint can be found at Line 161.

3.4 Constraining d-shaped sticks

d-shaped sticks were also fairly simple to constrain. First, referring again to Figure 2, it is possible to notice how segments 1, 4, 7 will never be lit by a d-shaped stick.
The second necessary constraint, forces segment 5 to be lit by a d-shaped stick if and only if segment 2 is. The same holds for segments 3 and 6.
To constrain the number of d-shaped sticks, the procedure is slightly different from the one for u-shaped ones. Since a d-shaped stick is made of two segments, it will need two `stick` variables to be equal to `isD`. To formalize this, for all i, j there shall be at most $2 * d$ cells of the array `isStick` such that `isStick[i,j]=isD`.
The implementation of these constraints can be found from Lines 166 to 175.

3.5 Constraining c-shaped sticks

Constraining c-shaped sticks was perhaps the hardest part of the whole project.
Starting with the easier parts, I had to instruct the solver that whenever a segment is lit by a c-shaped stick, there has to be another one also lit by a c-shaped stick such that it forms a corner with the former segment. This is implemented at Line 180, for each possible corner.
Then, analogously to the d-shaped sticks, the number of `isC` values in the array `isStick` has been bounded to $2 * c$.

The issues arising relatively to c-shaped sticks lay in the fact that, leaving things with only the constraints just described, they can overlap. A 7-segments display contains 8 possible corners, though, if we want to arrange some c-shaped sticks on it, we won't be able to fit more than 3 of them, but the solver does not know this yet so nothing will prevent it from arranging 8 of them on a single display.
To tackle this issue I used different constraints which can be found from Line 189 to Line 195.
First, I simply limited the number of segments having `isC` values to 6 for each display, that is, limiting to 3 c-shaped sticks. Then it is necessary to instruct the solver that if both segments 2 and 3 are lit by a c-shaped stick, they cannot rely on the same segment to form a corner, so both segments 1 and 4 must be lit by a c-shaped stick. Same holds for segments 5 and 6 with 4 and 7. This prevents the solver from building the digit '4' using 3 c-shaped sticks as shown in Figure 3.

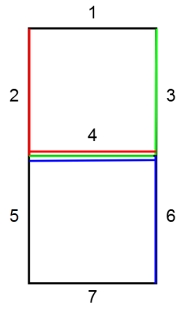


Figure 3: A '4' built with c-shaped sticks only.

This solves the problem from a vertical point of view but, horizontally, things are not quite done. For instance, it is now possible to build the digits '5' and '2' with 3 c-shaped sticks, exploiting, again, an overlap. This problem is solved by the constraint at Line 182 which does not allow an odd number of segments lit by a c-shaped stick.

One might argue that all these countermeasures are useless since, if 3 c-shaped sticks are available the solver will use them to build the digit '9' given its higher value, but I decided to take care of these issues anyway in order to get a model that was actually able to build all the digits.

3.6 Constraining l-shaped sticks

The way I modelled l-shaped sticks is very similar to the one I used for c-shaped sticks, just slightly easier.
Again, the main issue regards possible overlaps between different sticks. This is taken care by the constraint at Line 210 which does not allow the segments 2, 5 and 3, 6 to rely on the same segment (1 or 7) to form a corner.

Finally, Lines 216 and 218 take care of the number of segments lit by d-shaped sticks, firstly by limiting the number of `isL` values to 6 for each display in order to further prevent overlaps and then by bounding the number of these values to $3 * l$ for the whole grid (since each l-shaped stick requires three segments to be built).

3.7 Optimizing the result

In order to optimize the solutions by both the number of digits and the sum of the digits prioritizing the former, I decided to maximize the arithmetic expression:

$$digitsNumber * maximumSum + digitsSum$$

where $maximumSum$ is the maximum obtainable sum of digits given the number of displays (if they were all '9').
Such expression makes it so that the sum of the digits only becomes relevant to the result when the number of digits of two solutions is equal.

3.8 Breaking symmetries

Leaving the model as it is would have the solver checking for all possible permutations of digits, consisting in a very inefficient computation. To tackle this issue, initially, I forced the digits represented by the displays in a decreasing ordered (this part of the code can be found from Lines 137 to 151). This did indeed improve the performance, but it would still allow many more permutations of the sticks around the grid, especially u-shaped sticks (the least constrained ones).
Symmetry breaking constraints can be found from Lines 248 to 261 (commented out). Line 248 orders the digits based on the number of u-shaped sticks in them. Lines 251 to 261 force the solver not to use u-shaped sticks to form the same shapes of d, c and l-shaped sticks, if these are still available. According to my tests, this further improved

the performance for smaller instances, although for larger ones, the first solution seemed to be way better. Since both symmetry breaking constraints could not work together, I decided to keep only the one which avoids different permutations of the same digits. I dive a little deeper into this topic in Section 6

3.9 Search primitives

As for the search primitives, after thorough testing, I have found out that the best search strategy is:

```
int_search(isStick, input_order, indomain_median)
```

I think it is quite simple to justify these results:

- `isStick`: This is the variable around which the whole model goes around. Having an efficient search on this variable means having a huge positive influence on the rest of the computation.
- `input_order`: Choosing the variables in the array with `input_order` allows to choose them based on their position in the array. I think this annotation is promising because of the way the symmetries are being handled. By first determining the digits in the first displays it is possible to limit the search space for the following ones.
- `indomain_median`: Assigning to the variables of `isStick` the median of their domain means assigning them the value `isC`. What I concluded is that the computation is more efficient with this choice because *c*-shaped sticks are the more constrained ones, consequently allowing to quickly decide whether the value is good or not.

A valid alternative to `input_order` was `first_fail`, which should be more and more efficient the larger the instances, given the fact that in the beginning, variables have very similar domains.
For quite a while, I have also been testing search primitives which included randomness like `indomain_random`, complemented with `dom_w_deg` (in variable choosing) and with different `restart` policies. Although, since they were not available for all the solvers I stopped considering them. Also, I preferred using deterministic primitives since I could reason more on the causes that would give certain results.

4 My ASP solution

As stated in the beginning of this document, this part of the project was deeply inspired by the Minizinc model so the reasoning behind it is mainly the same.
With that in mind, I would still (not proudly) like to highlight how, even though I already

had the solution in mind, I struggled to transit from Minizinc to ASP. The difference between the two programming languages is thin but the change in paradigm requires to utilize a slightly different approach, which can be tricky to do, especially when defining the foundations of the model.

In order to give a better structure to both the code and the illustration I am following the conventional subdivision into the three main parts: *Generate, Define, Test, Display.*

4.1 Generate

Defining the solution candidates for the 7-segments problem means defining the possible arrangements of the sticks in the grid. To do this, I am using two different predicates: `isLit/2` and `isLit/3`.
`isLit/2` is used to express that a segment `S` of a display `D` is lit. `isLit/3` does the same but also takes care of specifying the kind of stick `ST` used to light up the segment. The rule at Line 15 is used to describe that whenever a segment is lit a stick is being used.
Initially I was only using `isLit/3` but I immediately noticed that, when defining the shapes of the digits, the type of stick was irrelevant, therefore I felt like using the second predicate `isLit/2` was a good idea. For this idea I was inspired by the Hanoi-Towers implementation in [1].

4.2 Define

My solution makes use of only one, very important auxiliary predicate: `hasDigit/2`. This predicate indicates the digit contained by a display `D`. `hasDigit/2` allowed me to define the shapes of the digits similarly to how I did in Minizinc (see Subsection 3.2).

4.3 Test

Finally, the *test* section of the model contains the constraints on the arrangements of the sticks. These constraints have been defined here in the same exact way as they were defined in Minizinc. For better clearance, an illustration of the main, more tricky parts follows.

4.3.1 *c*-shaped sticks

Constraints from Lines 139 to 145 are used to force an `isC` segment to have another `isC` with which it is able to form a corner. Line 148 avoids the problematic case of the digit '4' shown in Figure 3. Constraints from Lines 151 to 154 prevent lateral pairs of facing segments (2, 3) and (5, 6) to form a corner with the same segment (either segment 1, 4 or 7).

4.3.2 l-shaped sticks

Lines 157 to 160 makes it so that if one lateral segment (2, 3, 5, 6) is lit by a l-shaped stick, the one below or above it must be as well. Lines 162 and 163 force the lateral segments to have a third segment with which they can form a corner (either 1 or 7). Lines 165 and 166 prevent segments 1 and 7 to be lit by l-shaped sticks on their own. Finally, Line 168 prevent the two pairs of lateral adjacent segments ((2, 5) and (3, 6)) to use the same segment (either 1 or 7) to form a corner.

4.3.3 Maximization and symmetry breaking

In this case the arithmetic expression being maximized is exactly the same as the one in the Minizinc model. Symmetry breaking is also done analogously as in Minizinc (from Lines 179 to 181), although I would like to add that the predicate `hasDigit/2` really made this task feel simpler and on a higher level than it was for Minizinc.

4.4 Display

Line 184 makes it so that only the predicates `hasDigit/2`, `digitNum/1` `digitSum/1` `isLit/2`, `isLit/3` are shown.
I have also developed a python script `7-segments_ASP_pretty_printing.py` that allows to get a pretty printing of the solution. To use it, it is necessary to execute it and give as argument the n, u, c, d, l parameters. For instance calling from directory `ASP`:

```
python 7-segments_ASP_pretty_printing.py -n 2 -u 2 -c 3 -d 1 -l 0
```

Solves the problem with parameters $n = 2, u = 2, c = 3, d = 1, l = 0$.

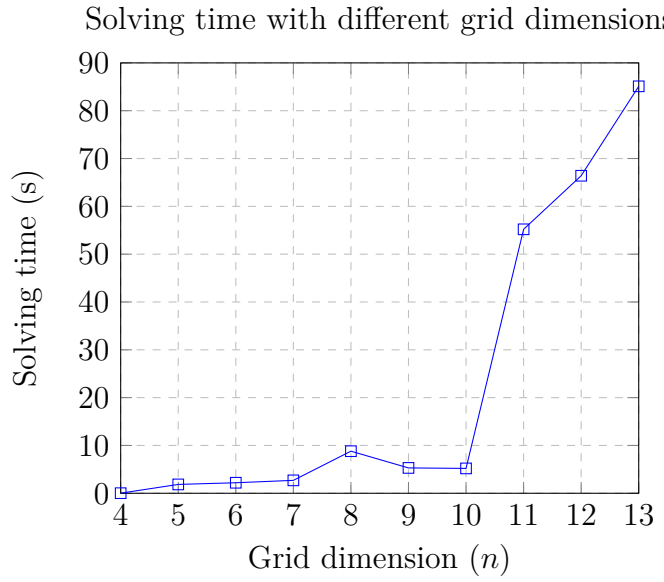
5 Testing

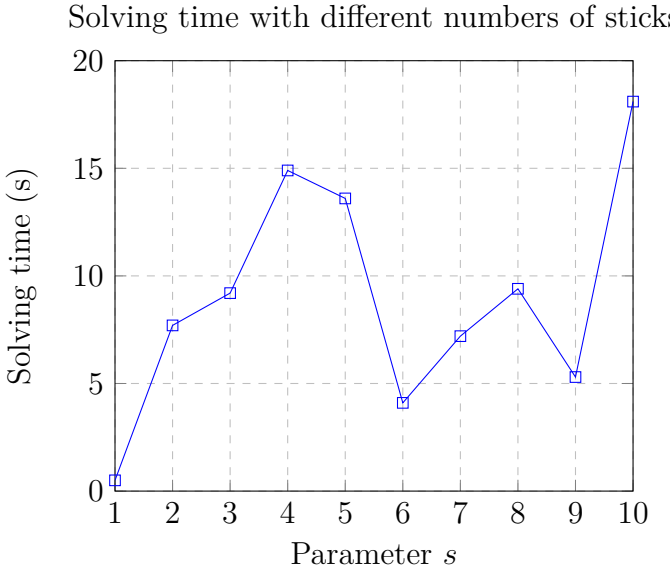
Following are the test results of the two models. Mind that, since the ASP model has much higher solving times, I have decided to test it on smaller instances. With that being said, there are still common dimension instances which will allow a comparison between the two.

5.1 Minizinc

As for the Minizinc model, the testing has been performed with the solver `COIN-BC` which seemed to be the most performing.
The model has been tested with grid dimensions from 4 to 13 (100 instances, 10 for each value of n). Meanwhile, the parameters u, c, d, l have been assigned for each instance a random value between $n - 3$ and n. The tests are launched from a python script which uses the Minizinc Python package. The script, the random generated instances and the

results can be found at path `Minizinc/testing/`.
The results plotted on the chart below show the average results of each of the 10 groups of instances. It is possible to see how, for smaller instances, the parameters u, c, d, l have as strong influence on the results as the parameter n. With larger instances, n becomes more and more decisive, given the quadratic growth of the maximum number of displays.





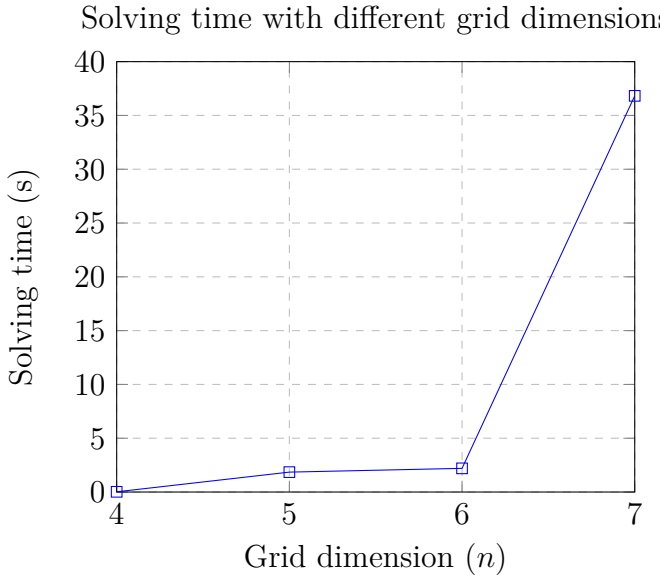
To make this point clearer, u, c, d, l do have influence for larger instances (with no constraints there would be 4^7 possible combinations for each display). The point, again, is that also this number is strictly related to the number of displays. To support this, I also tested the model fixing $n = 10$ and increasing the number of sticks (second chart

above). More specifically I used a parameter s, parameters u, c, d, l are assigned a random value between s and $s-1$. The test was done for 100 instances, 10 for each value of s, what is shown is the average of the groups of instances.

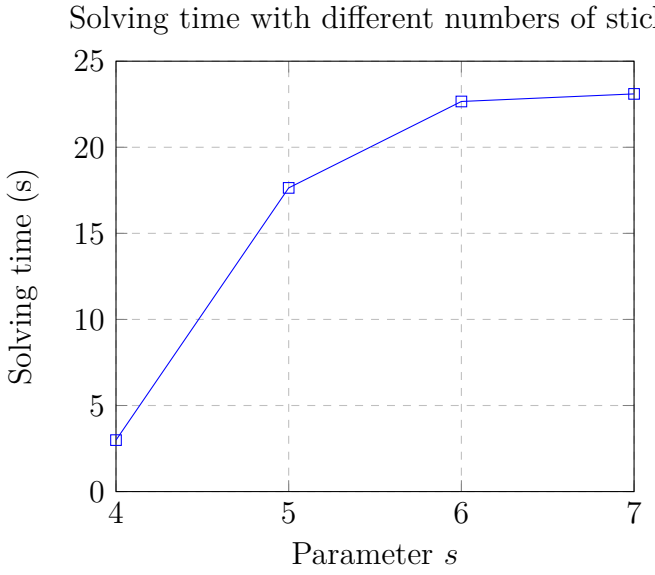
Lastly, even though I did not pay much attention to it, I also briefly analyzed the statistics regarding the constraints. I did not find it too interesting since obviously the statistic is strictly related to the number of displays.

5.2 ASP

As mentioned above, regarding the ASP model, I was only able to test for small instances, as for larger ones the solving times were just too long. For this model I used the solver on default settings, on 100 instances (25 for each value of n). Shown, again, are the averages of each group of instances.
The scripts, the random instances and the results can be found at path `ASP/testing`.
Here it is possible to see how the ASP model does keep up with Minizinc's performances on small instances, although the blow up on the growing of n is way larger. I did not plot this data in the chart, but with $n = 8$ the solving time was over 5 minutes.



For completion, also in this case I tested the model fixing $n = 7$ with varying u, c, d, l. This test was done on 32 instances 4 groups of 8 with a growing parameter s increasing from 4 to 7, where the parameters u, c, d, l were assigned a random value between $n-3$ and $n-1$. These results show how the number of sticks has a much stronger influence on the results with respect to the Minizinc model. I concluded that this could be caused by the search primitive that are being used in Minizinc, more specifically, I am referring to `indomain_median`. I think having a more efficient value assigning is what makes the number of sticks less relevant in Minizinc. The results can be seen in the following chart.



6 Conclusions and possible optimizations

Concluding, I think that developing the same problem with two different paradigms is what puts all the results into perspective. I find this problem very interesting given the amount of different variants it can expand into. For instance, I initially started developing a solution for non-fixed displays number (see Section 2), where the grid was manipulated as a directed graph. This approach ended up being very complex with high solving times and (ironically) way less interesting solutions.
Focussing on the solution I have delivered, I think it could be optimized by deeply analising symmetries in equal solutions with different arrangements of sticks. For instance, with the model as it is, given $n = 2, u = 6, c = 3, d = 0, l = 0$ there are still 8 equal solutions with different arrangements of the u and c-shaped sticks to form two digits 9. Finding a solution to this would largely improve performances, although, from what I noticed, the difficult part is to be able to guarantee no loss in generality.

References

[1] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, S. Thiele, and P. Wanko, *Potassco User Guide*, pp. 12–16. 2019.