Solving mathematical exercises through the use of constraint satisfaction and logic languages

Report for the project work of the course
"Languages and Algorithms for AI - Module 1"

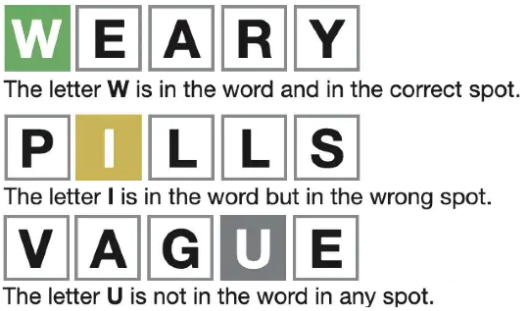
Biagini Diego
Pittini Enrico
Simeoni Ildebrando

1 Wordle

1.1 The game of Worlde

Wordle[1] is a simple word guessing game which has become popular lately on the internet.
The rules are as follows:

- the player has to guess a single unknown 5 letter word
- only 6 guesses are allowed at most
- each time the player makes a guess and for each letter in it, the game gives them some information on the hidden word based on how close the guess was to it



1.2 Solving the game

The game environment can be represented easily as a list of strings, each one of them being one of the already made guesses.
The solution instead will be a single ordered list of letters.
We then have to represent the hints the game gives us for each guess and for each letter in the guess. This is done through a $NGuesses \times 5$ array, composed of identifiers which tell us whether the corresponding letter was in the word or not and whether it was in the correct position or not. In our implementation we used the following notation:

Using this array we can constrain the solution components to be equal or not to the ones in the guesses. Thus the 3 main constraints are:

- if the hint is a 0 the corresponding letter is not part of the solution
- if the hint is a 1 the corresponding letter is part of the solution
- if the hint is a 2 a certain letter in the solution takes the value of that letter

Furthermore our solution has to be a valid word in the English language, for this reason we have to add a dictionary of words into our programs from which to pull words.

[1] `https://www.nytimes.com/games/wordle/index.html`

1.2.1 Minizinc implementation

Due to some technical restrictions on minizinc string variables, it's preferable to represent letters in the solution, guesses and dictionary as integers in a known range (1 to 26) rather than single characters.
The dictionary can be easily implemented as an array of integers appropriately saved in a .dzn file, this way we can quickly swap it with a simpler or more complex one as needed. To force the solution to belong to the dictionary we say that there exists a index for which the solution is equal to the word registered in the dictionary at that index.
With regards to the 0 and 1 hints we can use the set data structure to force the needed constraints, that is we define an unordered set from the solution(which is ordered and might have repeating digits) and say that a certain letter is or isn't in this set.

1.2.2 Prolog CLP implementation

In Prolog strings are better handled than Minizinc, so we can represent our guesses as a list of strings and turn individual letters back and forth between their ascii representation and an integer in the 1..26 range.
The dictionary is represented here as an array of integers as well, however the constraint that the solution must belong to the dictionary is much more naturally implemented by using the $member(A, X)$ predicate.
The constraint implementation is straightforward, the hint array is scanned and each time we find a 0 or 1 we force an inequality constraint; if instead a 2 is found we force and equality one.
In addition to this, for each 1 hint we use the *member* predicate to force a letter to belong to the solution.

1.3 Which are the best guesses to make?

Finding a solution given a set of guesses is good and all, however the guesses have to be good to restrict our possible solutions to only a single one, which should end up being the hidden word.
We can set up a COP to find this set of optimal guesses.
This boils down to finding the set of words, for which the cardinality is known, which:

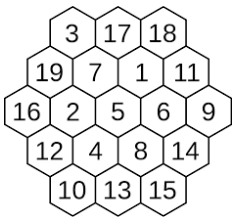
- has the highest number of different characters between words in the set
- if it's not possible for the set to contain all characters then it should only contain the most frequent letters

To find out which are the most frequent letters we took a list containing a lot (thirteen thousand) of 5 letter words and we computed the absolute frequencies of each letter.

1.3.1 Minizinc implementation

The main parameter set by the user is the number of words we want to find. The decision variables are an array of indexes which correspond to a word in the dictionary we used for the first problem as well.
We can address the dictionary to extract the chosen words and put them in a set to find out all the different letters, the first term of the formula to optimize is the cardinality of this set. Then we compute a score for the set, as a sum of the relative frequencies of the letters in the set, this is the second term of the formula.
Several improvements can be made to optimize the search, for example by forcing symmetry breaking constraints, turning the function to optimize from a real valued one into an integer valued one and choosing a different search strategy than the default.

2 Magic Hexagon



2.1 Problem

A magic hexagon of order n is an arrangement of close-packed hexagons containing the numbers 1, 2, ..., $H_(n-1)$, where H_n is the nth hex number such that the numbers along each straight line add up to the same sum. It turns out that normal magic hexagons exist only for n = 1 (which is trivial, as it is composed of only 1 cell) and n = 3. In the above magic hexagon of order n=3, each line (those of lengths 3, 4, and 5) adds up to its magic number: 38.
In 1963, Charles Trigg proved by mathematical analysis that the order 3 hexagon is the only magic hexagon of any size disregarding rotations and reflections.

2.2 Minizinc implementation

The parameters are: the integer range in which the elements of the magic hexagon of order 3 belong to, the integer magic number which represents the sum of all elements of a line and all the elements of the hexagon which is represented as a 1D array of dimension the hex number.
The constraint on the uniqueness of all the elements of the hexagon is taken into account by the all different global constraint, while the constraint on the sum of all the elements at each straight direction being equal to the magic number is dealt with three constraints on the sum of the elements, one for each direction in which the sum can be done.(i.e. horizontal, main diagonal and anti diagonal direction).
For better visualization the output of the program (i.e. the values assumed by the elements of the magic hexagon) is shown both in an array style and in an hexagon shape.

2.3 Prolog CLP implementation

In Prolog two solutions are presented, in particular the first one recreate the Minizinc solution by defining the Magic Hexagon as a 1D array of variables in the integer range 1..19 and checks the uniqueness of all the elements of the hexagon and the constraint on the sum of every line by specifying each possible sum that must be equal to the magic number.
An alternative and more compact solution is provided, this one makes use of the predicate "maplist" thanks to which we are able to apply the user defined predicate "sum magic" to all the possible sublists of elements of the hexagon, which ensure the check of the sum constraint.
The query "magic hexagon(Hex), labeling([leftmost], Hex)." allows to find the elements of the hexagon and label them (in a leftmost option) assigning a value to each variable in Hex systematically trying out values for the finite domain previously defined.

3 Athletics committee problem

3.1 Problem

The problem, taken from Peter Wrinkler's Gathering 4 Gardner convention states:
"Every professor wants to be on the Athletics Committee – free tickets to your favorite event! To keep the committee from becoming too cliquish, the college forbids anyone from serving who has 3 or more friends on the committee. That's OK, because if you have 3 or more friends on the committee, you get free tickets to your favorite event! Can the committee be chosen so that no one on it has 3 friends on the committee, but everyone off the committee has 3 friends on it?"

3.2 Assumptions

- Friendship is a symmetric relation.
- A person is not a friend of himself.
- Everyone off the committee has exactly 3 friends in it.
- There must be at least one person in the committee and one outside it.
- To let the committee being not too crowded it must have less than half of number of total professors in it.

3.3 Minizinc implementation

The parameters are: the total number of professors, a 2D array that can track the friendship relation between professors and a 1D array that tracks the membership of every professor to the committee. Constraints on the number of people in the committee are solved by checking the sum of the 1D array of memberships, while constraints on the number of friends in the committee are dealt with a weighted sum of all friends of each professor. Assumptions on predicate friendship are later dealt by checking its symmetry and not reflexive properties.
The problem can be seen both as a CSP problem by looking at the configuration that satisfies the constraints above mentioned and as a COP problem by trying for example to minimize the number of committee members or maximizing it.
Being n the number of professors for $n < 4$ the CSP problem as we defined it is unsatisfiable, while for increasing values of n the number of possible solutions grows exponentially. For this reason the brute force approach of starting from a random committee and then refine it by solving the constraints (generate and test approach) would be infeasible.
The output shows for each professor its membership to the committee (1 member, 0 not member) and all his friends, at the end also a list of the entire committee is shown.

4 Autonomous numbers problem

An autonomous number is a natural number N in which the digit 0 is not present and which satisfies the following property: counting the number of times each digit is used, from the smallest digit to the biggest digit, the number N is itself found, from left to right.
Example
21322316 *is an autonomous number. The following digits are used: two* 1 *; three* 2 *; two* 3 *; one* 6*. Appending these results, the number* 21322316 *is found, left to right.*

The aim of the problem is to find an autonomous number. This search is filtered with additional optional constraints, involving the number of digits of the number and the divisibility of the number. For example, the user can specify that the autonomous number N must have a number of digits between 8 and 17 and that it must be divisible by 11 and 7 but not by 2 and 9. (In other words, N must have 11 and 7 as divisors but not 2 and 9).
In the CLP implementation, the user can ask about the divisibility of the autonomous N number by any number. Instead, in the MiniZinc implementation, the user can ask about the divisibility by only numbers in the range 2..11. (The reason of this limitation will be explained later on).

4.1 Solution

The problem has been solved using both MiniZinc and CLP Prolog. In both cases, the idea of the solution is very similar.
Counting the indices of the digits of the number from left to right, starting from 1, the following constraints must be ensured.

- The number of digits must be in the range specified by the user.
- All the digits must be different from 0.
- The number of digits must be even.
- The digits with even index must be in a strict ascending order.
  Example: the number 763849 *is compliant with the constraint, while* 763649 *is not (because* 6 *is not strictly less than* 6*).*
- For each digit with even index, the previous digit in the number must represent the count of that digit in the number.
  Example: the number 311415 *is compliant with the constraint, while* 312415 *is not, because there aren't two digit* 4 *in the number (and there aren't three digit* 1*).*
- Each digit with an odd index must be equal to at least one digit with an even index.
  Example: the number 21322314 *is compliant with the constraint, while* 311415 *is not, because the digit* 3 *is not present in any even-index digit.*
- The autonomous number must be divisible by some numbers specified by the user and it must be not divisible by some other numbers specified by the user.

4.2 Implementation details

Regarding the MiniZinc implementation, the autonomous number is not represented as a number, but as an array of digits. This is done due to the integer overflow: in MiniZinc an operation between integer variables overflows if it exceeds 10^{11}. This is a big limitation for our problem (the biggest autonomous number has 18 digits).
This is the reason why the user can't specify a generic divisor, but only divisors in the range 2..11: indeed, for the numbers in this range , there are divisibility rules which can be applied on the array of digits, rather than on the actual number.

Instead, in the Prolog implementation, the autonomous number is represented as a number, even if also the representation as an array of digits is internally used, for simplifying the check of some constraints.
Therefore, the user can specify any divisor.

5 Barrels problem

A grid of barrels is given: grid $r_{barrels}$ **x** $c_{barrels}$. The total number of barrels is $r_{barrels} * c_{barrels}$.
Each barrel is a grid of cells: grid $r_{perBarrel}$ **x** $c_{perBarrel}$.
Therefore, on the whole, we have a grid of cells which is $(r_{barrels} * r_{perBarrel})$ **x** $(c_{barrels} * c_{perBarrel})$.
Each cell contains a number, which is in the range $1..m$.
Each barrel has associated a number, which is the sum of all the elements in that barrel: this number is put on the top of the corresponding barrel.

Example with $r_{barrels} = 3, c_{barrels} = 4, r_{perBarrel} = 2, c_{perBarrel} = 3, m = 14$.

Barrel 0	Barrel 1	Barrel 2	Barrel 3
Sum 46	Sum 45	Sum 31	Sum 35
[13, 8, 2]	[5, 6, 10]	[3, 1, 7]	[9, 4, 11]
[6, 12, 5]	[4, 9, 11]	[8, 2, 10]	[7, 1, 3]
Barrel 4	Barrel 5	Barrel 6	Barrel 7
Sum 43	Sum 41	Sum 34	Sum 38
[5, 2, 11]	[7, 3, 4]	[1, 12, 9]	[10, 8, 6]
[8, 10, 7]	[6, 12, 9]	[5, 4, 3]	[1, 11, 2]
Barrel 8	Barrel 9	Barrel 10	Barrel 11
Sum 49	Sum 37	Sum 44	Sum 26
[7, 6, 3]	[10, 2, 1]	[12, 9, 8]	[11, 5, 4]
[12, 11, 10]	[9, 8, 7]	[6, 5, 4]	[3, 2, 1]

On the whole, we have a grid of cells with size 6 x 12.

In building such structure, there are the following constraints: each row of the grid of cells must contain all different numbers; each column of the grid of cells must contain all different numbers; each barrel must contain all different numbers; all the numbers on top of the barrels must be different numbers.
In addition, the user can specify that some specific numbers must be present on top of the barrels.

5.1 Solution

The problem has been solved using both MiniZinc and CLP Prolog. In both cases, the solution is very similar.
The two main entities are: the overall grid of cells, which is a matrix (in Prolog a list of lists); and the array/list of numbers on top of the barrels.
Each element of the grid of cells must be a number in the range $1..m$.
Each row and each column of the grid must have all different elements: this is a classic and easy check to perform, in both languages.
More difficult is the check of the constraints involving the barrels, which are two: each barrel must contain all different elements; each number on top of a barrel must be the sum of all the elements in that barrel. The check of these constraints is based on building the barrels from the grid of cells, which means extracting the submatrices representing the barrels.
Then, the numbers on top of the barrels must be all different.
Finally, all the numbers specified by the user, if any, must be present on top of the barrels.