Constraint Logic Programming approach to White and Tan Puzzles

Pedro Seixas[up201806227] and Telmo Baptista[up201806554]
from 3MIEIC03, White and Tan_2

Faculty of Engineering of the University of Porto
https://sigarra.up.pt/feup/en/web_page.inicial

Abstract. This paper presents a Constraint Logic Programming approach to solve and generate White and Tan puzzles using SICStus. This method showed to be very helpful when solving these kind of puzzles. The puzzle rules were simple to implement as constraints and the program can efficiently solve and generate puzzles of various sizes.

Keywords: Prolog · Logic Programming · Constraint Logic Programming · clpfd · Puzzle · White and Tan

1 Introduction

This paper was proposed in the Logic Programming course with the main intent to make use of the Constraint Logic Programming to solve puzzles in a more efficient and robust way. Our main goal was to, not only solve these kinds of puzzles, but also to generate valid ones.

We will be starting with the problem description where we explain these types of puzzles and how to solve them. After that, we show our approach to solving this problems, with both the decision variables and the constraints used. Afterwards, we present the tests made to our program, followed by some conclusions and future work.

2 Problem Description

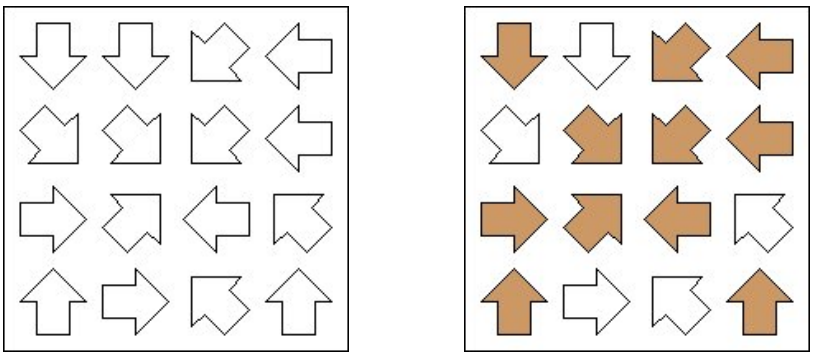
Given a `NxN` matrix of arrows, the puzzle is completed by colouring some of the arrows so that:

- 1. Each white arrow points exactly to only 1 white arrow
- 2. Each tan arrow points exactly to 2 tan arrows

With this we can already eliminate all matrix that are 2x2 or 3x3 due to being impossible or not relevant as:

- 1. In a 2x2 matrix, all the arrows would need to be white, as an arrow can't ever point to two arrows. With the restriction of all arrows being white this puzzle doesn't make sense on this type of matrix.

– 2. In a 3x3 matrix, if an arrow is tan then it needs to point to two arrows, so by snowball effect an arrow at the middle of a column or row needs to be tan, thus the middle cell of the matrix will always need to be tan but an arrow in the middle cell of the 3x3 matrix can't ever point to two arrows, resulting in an impossible puzzle.



(a) Puzzle (b) Solution

Fig. 1: Example

3 Approach

We took a Constraints Logic Programming approach in order to solve the puzzle *White and Tan* using the SICStus library clp(fd) - *Constraint Logic Programming over Finite Domains*.

After some failed implementations, we finally came up with one that we could use in our solver and that would be user friendly, i.e. that everyone could understand easily what the values meant. In this implementation, the solver takes a NxN matrix of directions, representing the directions of the puzzle's arrows, where each direction is one of the following: *w, nw, n, ne, e, se, s, sw*, representing the 8 directions pointed by the cardinal points.

3.1 Decision Variables

The result is a matrix the same size as the input where each value is either 0 or 1, whether the arrow is tan or not (1 meaning the arrow is tan).

3.2 Constraints

For the input, it is checked if the input matrix is a square matrix where each side is bigger than 3, as there aren't any puzzles smaller than 4x4. As for the

puzzle restrictions, the only restrictions used are the ones that are described in the puzzle description. If the arrow is white, then it must point to only one white arrow. Otherwise, it must point to exactly two tan arrows.

For these constraints, the predicate **direction_arrows(+Position, +Matrix, +DirectionOffset, -DirectionArrows)** was needed in order to get the elements of a matrix in a given direction. To apply the constraints, we used the *clpfd* predicate **global_cardinality/2** to get the number of white and tan arrows in each arrow direction and applied the restriction that ensured that if the arrow was tan, then it should have 2 tan arrows in the direction arrows list, otherwise it should have 1 white arrow, this was achieved by the following block of code:

$$
\begin{aligned}
&direction(Direction, OffsetX, OffsetY),\\
&direction\_arrows(X-Y, Solution, OffsetX-OffsetY, DirArrows),\\
&global\_cardinality(DirArrows, [0-N0, 1-N1]),\\
&((Arrow\# = 1\# \wedge N1\# = 2)\# \vee (Arrow\# = 0\# \wedge N0\# = 1))
\end{aligned}
$$

To generate a puzzle, it's used the predicate **solutionConstrains(+Arrows, +Position, -SolutionMatrix)** that applies the constraints to the solution being generated with the predicate **applyRestrictionsDirections(+Arrow, +Position, -SolutionMatrix)**. The later applies restrictions to each direction the arrow at position X, Y can point, guaranteeing that at least one direction makes that arrow follow the puzzle rules by using the predicate **applyRestrictions(+Directions, +Arrow, +Position, -SolutionMatrix, -N)**, being N the number of directions that obey the game rules restriction, and to generate a solution N must be greater than zero, thus in the predicate **applyRestrictionsDirections** it is applied the restriction `Restrictions #>0`, being `Restrictions` the value returned in N, a short snippet of code is presented below representing the **applyRestrictions** predicate:

$$
\begin{aligned}
&direction_arrows(X-Y, SolutionMatrix, OffsetX-OffsetY, DirArrows),\\
&global\_cardinality(DirArrows, [0-N0, 1-N1]),\\
&((Arrow\# = 1\# \wedge N1\# = 2)\# \vee (Arrow\# = 0\# \wedge N0\# = 1))\# <=> B,\\
&N\# = M + B,\\
&applyRestrictions(Rest, Arrow, X-Y, SolutionMatrix, M).
\end{aligned}
$$

4 Solution Presentation

There are four main predicates to get the solution of a problem. The first two print the text and the visual representation and the last two print only in text mode. The visual representation is achieved by printing Unicode arrow symbols that represent each direction and its color. All of them besides the last one take as arguments both the problem, which dictates the arrow directions, and the solution. Only the last one, which only prints the text solution, doesn't need the problem.

The first and the third, **print_ps/2** and **text_ps/2**, where *ps* means Problem and Solution, print the problem followed by the solution whereas the second and fourth, **print_sol/2** and **text_sol/2**, print only the solution.

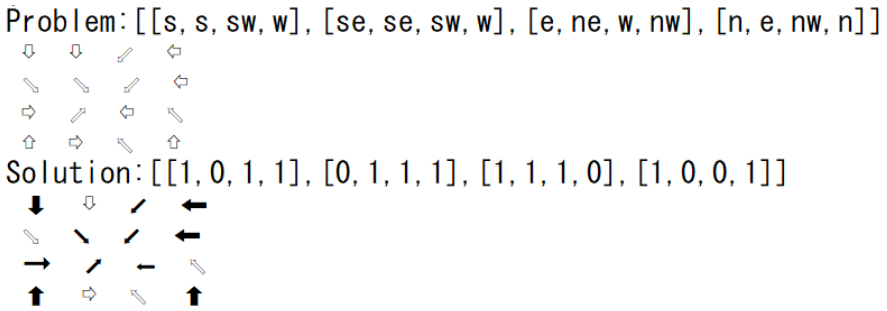


Fig. 2: Problem and Solution presentation using both the text and the visual representation, with predicate **print_ps/2**

For user interaction, the predicate **menu/0** can be called which will display a menu with an option to choose an ID of a puzzle listed in the file **problems.pl** that include the already generated puzzles via the predicate **generate_puzzle** and then given the option to show the solution.

The arrows alignment isn't perfect due to the available fonts in SICStus not accepting mono space Unicode symbols. Anyways, it is fairly simple to understand the visual representation, since the misalignment is barely noticeable.

5 Experiments and Results

There where made experiments and tests to both our solver predicate and our generator predicate in order to compare execution times and put together the most efficient methods.

5.1 Dimensional Analysis

With the dimensional tests we intended to see the problem resolution time depending on the size of the puzzle. The following graph shows the increase of time it took to solve 100 problems from puzzles 4x4 to 10x10. We couldn't test with 11x11 or bigger puzzles since our generator either took too long or did not found at all any puzzles with that size.

For the labeling, we used **ff** for the variable selection order, **median** for the way in which choices are made and **up** for the order in which the choices are made which showed to be the best heuristics combination for our problem (as demonstrated in the next section).

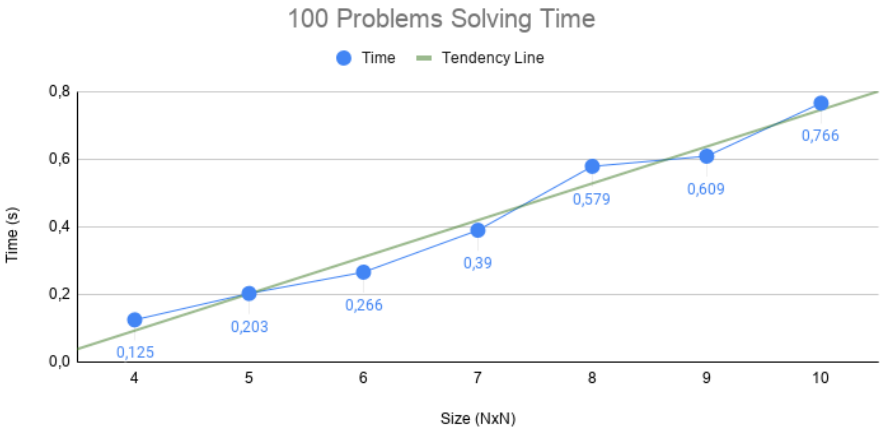


Fig. 3: Time to solve 100 NxN problems

This graph shows a linear increase in the time spent solving the puzzles when increasing the size of the puzzle.

In order to test the solver, we needed to use our generator to get the problems. The time it took to generate and solve 100 problems was measured and it is presented in the table below.

Size (NxN)	Time (s)
4	0.391
5	0.766
6	1.343
7	1.985
8	4.015
9	15.375
10	3076.375

Table 1: Generating and Solving 100 problems

As we can see by this table, generating problems gets a lot more time consuming as the size increases, having a significant increase with problems 10x10. This table shows an exponential growth when increasing the size of the puzzles to be generated.

We were surprised to see the difference between the generator and the solver, since the solver tests did not show any substantial increase in 10x10 puzzles, as the generator did.

5.2 Search Strategies

For the search strategy, we started by trying to select a random variable to start labeling which later demonstrated to be a lot more time consuming than the default values.

After that, we started testing with other heuristics and the results were very similar so we ended up testing all the combinations to see if any of them would turn out a surprise on both the positive and negative sides.

As we can see from the graph in Figure 4, neither of them had a significant time increase or decrease since the biggest time difference between two heuristics is less than 2 seconds.

From this graph we concluded that the best heuristics for our solver were **ff** for the variable selection order, **median** for the way in which choices are made and **up** for the order in which the choices are made which took only 2.86s to solve 700 problems of increasing size. On the other hand, the worst result was obtained when the variable selection order was changed to **anti_first_fail**, which took 4.031 seconds to solve the problems.

6 Conclusions and Future Work

This project was very challenging at first since we weren't used to work with constraints but quickly became very easy to understand and to develop. The use of Constraint Logic Programming made this project a lot more interesting and helped us understand the power that can be obtained with these type of programming.

We are happy with our results because our solver did not had an exponential increase in time when increasing the puzzle size as we initially expected. Higher volume testing can also be left for future work since we made tests with only a small portion of the White and Tan puzzles that exist.

Our generator was able to generate puzzles very quickly for small sizes but when the size was increased it was a lot more time consuming, which prevented us from testing with a bigger puzzle set.

To conclude, we think that the solver approach turned out to be a lot more simple than expected considering that we only needed to implement the puzzle rules as constraints.

References

1. Mats Carlsson et al: *SICStus Prolog User's Manual*. 4.6.0. RISE Research Institutes of Sweden AB, Sweden (April 2020)
2. *White and Tan Puzzle*, `https://erich-friedman.github.io/puzzle/whitetan/`

7 Annex

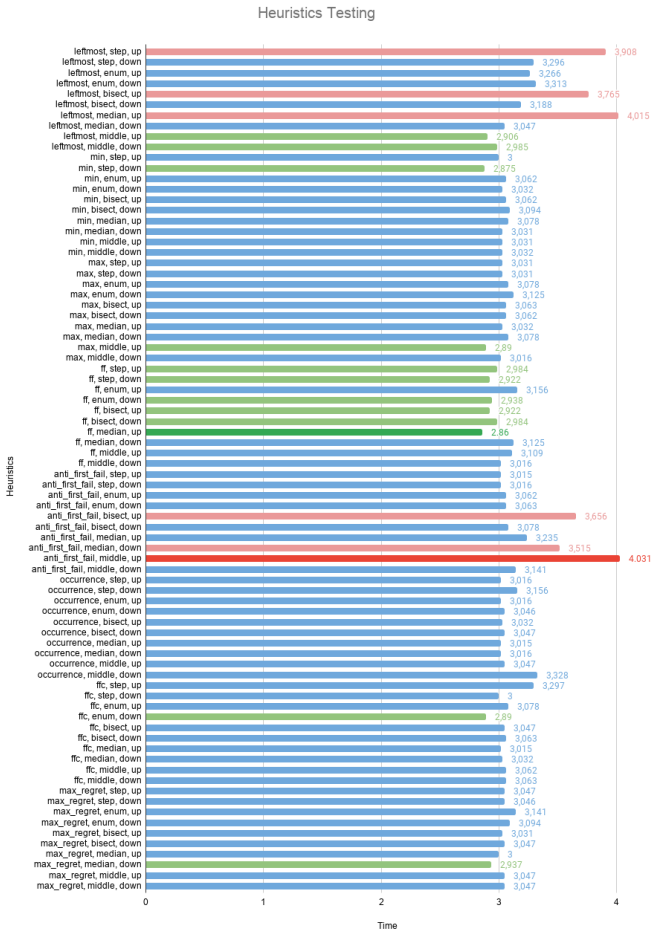


Fig. 4: Time to solve 100 problems of each size from 4x4 to 10x10 (total of 700 problems) with different heuristics combinations