Tricky Triple

António Dantas and João Cunha

FEUP-PLOG, Class 3MIEIC01, Group Tricky Triple-5
up201703878@fe.up.pt
up201403343@fe.up.pt

Abstract. This paper was written for the Logic Programming curricular unit as part of the second practical project. The main goal of this project was to solve a constraint satisfaction problem using the SICStus development system and applying our theoretical knowledge on Prolog's constraint programming interface. The chosen problem is a puzzle called Tricky Triple, where we must fill a board making sure that three adjacent white squares in the board contain exactly two of the three possible symbols. Our program has a database of predefined puzzles and predicates to display them. We were able to find a good global constraint predicate to solve the puzzles and with reasonable execution times, as we will explain in detail in this paper.

Keywords: Tricky Triple · Constraint Logic Programming · Prolog · SICStus.

1 Introduction

This project was developed for the Logic Programming curricular unit from the Faculty of Engineering of the University of Porto (**FEUP**). The goal of the project was to solve a constraint driven problem using Constraint Logic Programming (**CLP**). The problem we chose to explore was the **Tricky Triple** puzzle from *Erich Friedman.* In the next chapter the puzzle rules are explained in detail. It is known that a puzzle is just a set of rules that guide us to a unique solution. As such, it is possible to understand that this set of rules are the constraints of the given problem and that these puzzles are a great example of a Constraint Satisfaction Problem (**CSP**). A **CSP** is a problem that given a set of values that, when assigned to each variable, will find a solution the is able to satisfy all the constraints involved. On the next section we will explain in the detail the rules of the puzzle. On the following sections, 3 and 4, we will cover how we programmed the solver and the details of our implementation. Lastly, in section 5, we will give an analysis on our results and their efficiency.

2 Problem Description

As mentioned before, we are working with a puzzle named **Tricky Triple**, created by *Erich Friedman.* This puzzle consists of a squared board that can go from 4x4 up to 6x6, where each blank cell must be filled with either a triangle, a square or a circle. Some of the higher difficulty puzzles can also have black cells, that are considered off-limits and are not filled with any symbol, neither they count towards the 3 blank cells in a line. In our implementation of this puzzle, for simplicity reasons, the triangle, the square and the circle are represented by **1**, **2** and **3**, respectively, and the black cells are represented with a **0**. The board starts off with some cells already filled with a symbol. The goal is to fill every empty cell such that all three adjacent blank cells in a line (horizontally, vertically, or diagonally) contain exactly two of the symbols. Consider the following example:

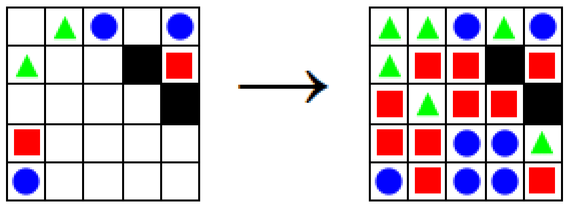


Fig. 1. Puzzle example with solution.

3 Approach

To develop our project we used the **PROLOG** programming language and the **CLPFD** library that gave use valuable operators and constraint predicates. Usually, a **CSP** (Constraint Satisfaction Problem) is modeled by:

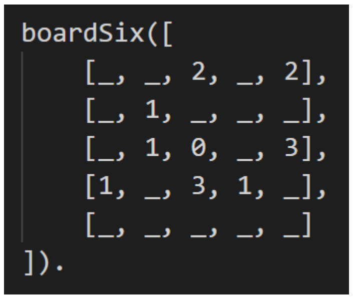
1. **Decision Variables**: representing different aspects of the problem alongside their domains.
2. **Constraints**: limiting the values the Variables can take.

The **solution** is achieved when each **Variable** has been assigned a value that satisfies all **Restrictions**.

3.1 Decision Variables

In our case, each cell of the puzzle will represent a variable. We thought this would be the best approach since every square is subject to comparisons, interactions with other cells and by the end of the run the board will have to be completely filled up. Their domains will be between 0 and 3. A **0** represents a blocked cell, **1** represents a triangle, **2** means its a square and **3** represents a circle.

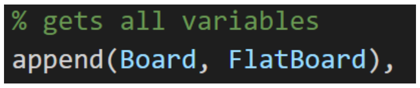
To achieve this, we implemented a board's database. It is a file that holds our test boards, they range from **4x4** to **6x6** boards. Each of them is a list of lists, containing blank variables (that will be filled by the constraints) and some cells that are already filled up.



```
boardSix([
    [_, _, 2, _, 2],
    [_, 1, _, _, _],
    [_, 1, 0, _, 3],
    [1, _, 3, 1, _],
    [_, _, _, _, _]
]).
```

Fig. 2. Board representation.

To easily access every variable in the board we then turn this into a single list containing all cells.

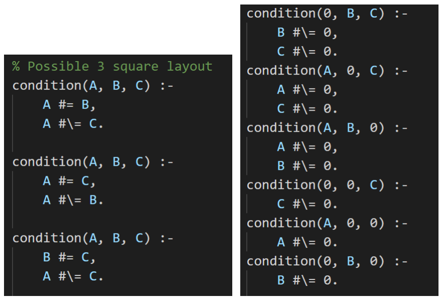


```
% gets all variables
append(Board, FlatBoard),
```

Fig. 3. Flat Board that will then be iterated.

3.2 Constraints

The constraints in our project are simple. When three adjacent white squares are in a line horizontally, vertically, or diagonally, they should contain exactly 2 of one of the symbols. A block cell can operate as any symbol. This means we will have to check every possible set of three cells in all possible directions. To achieve this we have created a predicate that restricts the values of their three possible arguments.

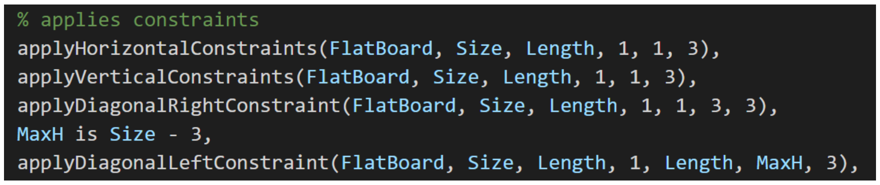


```
% Possible 3 square layout
condition(A, B, C) :-
    A #= B,
    A #\= C.

condition(A, B, C) :-
    A #= C,
    A #\= B.

condition(A, B, C) :-
    B #= C,
    A #\= C.
```

```
condition(0, B, C) :-
    B #\= 0,
    C #\= 0.
condition(A, 0, C) :-
    A #\= 0,
    C #\= 0.
condition(A, B, 0) :-
    A #\= 0,
    B #\= 0.
condition(0, 0, C) :-
    C #\= 0.
condition(A, 0, 0) :-
    A #\= 0.
condition(0, B, 0) :-
    B #\= 0.
```

Fig. 4. All possible Conditions.

In this way, we only have to iterate through the board, collecting all possible condition sets and applying that restriction to them. The program first inspects the board horizontally, then vertically, and finally it finds all their right and left diagonals. Each of these methods receives the flat board and row and column indexes to pin point exactly which piece is being analysed. It then compares the next cell it is going to iterate with a `Max` value that represents the last square of the three piece set. If `Max` exceeds the board's length it then passes on to the new row or column.

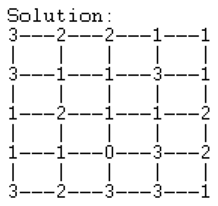


```
% applies constraints
applyHorizontalConstraints(FlatBoard, Size, Length, 1, 1, 3),
applyVerticalConstraints(FlatBoard, Size, Length, 1, 1, 3),
applyDiagonalRightConstraint(FlatBoard, Size, Length, 1, 1, 3, 3),
MaxH is Size - 3,
applyDiagonalLeftConstraint(FlatBoard, Size, Length, 1, Length, MaxH, 3),
```

Fig. 5. Applying constraints in all directions.

4 Solution Presentation

In order to make our puzzle solutions readable we have implemented a method that could turn a variable's list into something the user could understand. The `displaySolution` method receives a flat board and row and column's indexes, it then displays the board accordingly. It uses patches of `---` and `|` to separate each symbol and form a board.



```
Solution:
3---2---2---1---1
|   |   |   |   |
3---1---1---3---1
|   |   |   |   |
1---2---1---1---2
|   |   |   |   |
1---1---0---3---2
|   |   |   |   |
3---2---3---3---1
```

Fig. 6. Displayed Solution.

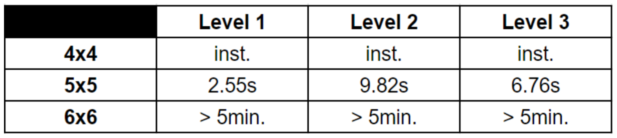
It uses a similar approach to the one given in the `applyConstraints` methods described above.

5 Experiments and Results

In order to test the efficiency of our implementation we conveyed some experiments to understand the range of its puzzle solving capacities. This will help us find and understand the best dimension and value selection heuristics to use in the labeling process.

5.1 Dimensional Analysis

Here we analyse the ways that different size puzzles influence the solving time. We have made it possible for our implementation to solve any board, but in this demo we will only use **4x4**, **5x5** and **6x6** puzzles. Here are the results.



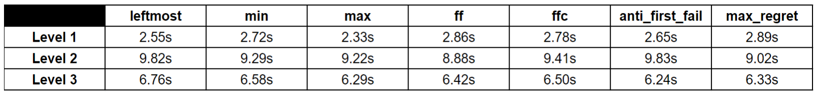
	Level 1	**Level 2**	**Level 3**
4x4	inst.	inst.	inst.
5x5	2.55s	9.82s	6.76s
6x6	> 5min.	> 5min.	> 5min.

Fig. 7. Dimensional Experiments.

We can see that the solving time increases exponentially as the board size increases. The smaller boards get solved instantly, whilst the bigger one's take a lot of minutes to find a solution. The **5x5** boards take only a few seconds and their differences in solving time depend on the puzzle's difficulty.

5.2 Search Strategy

To compare and find the best searching strategy we will try out different labeling methods. We will use the **5x5** boards as case study to run the project with every search mechanism.



	leftmost	min	max	ff	ffc	anti_first_fail	max_regret
Level 1	2.55s	2.72s	2.33s	2.86s	2.78s	2.65s	2.89s
Level 2	9.82s	9.29s	9.22s	8.88s	9.41s	9.83s	9.02s
Level 3	6.76s	6.58s	6.29s	6.42s	6.50s	6.24s	6.33s

Fig. 8. Search Experiments.

As we can see, the different labeling options don't seem to affect our results in a major way. Although, it is important to note that the best labeling option for Level 1 is `max`, for Level 2 `ff` got the best result and for Level 3 `anti_first_fail` was the best option.

6 Conclusions and Future Work

In conclusion, by working on this project we have learned how useful and powerful using Constraint Logic Programming can be to solve these types of problems. It required a lot of testing, failing, and learning to fully understand how to work with these tools, however it shows that once you understand and correctly implement them, it can offer you a variety of ways to reach the solution more efficiently. With more time it would be possible to implement a better and more efficient solution for higher difficulty problems as it was seen the efficiency dropped exponentially, the bigger the complexity of the puzzle. In sum, the project was concluded with success since we were able to implement the puzzle as a CSP with a valid solution.

References

1. John Charnley and Simon Colton. Expressing General Problems as CSPs.
2. M. Carlsson and T. Frhwirth, SICStus Prolog Users Manual. SICS Swedish ICT AB, 4.3.5 ed., 2016.
3. Tricky Triple Puzzle Page, https://erich-friedman.github.io/puzzle/shape/. Last accessed 4 Jan 2021
4. Constraint Logic Programming over Finite Domains, https://swi-prolog.org/pldoc/doc/_SWI_/library/clp/clpfd.pl. Last accessed 2 Jan 2021