Multi Balance Puzzle

Ricardo Fontão and Tiago Silva

FEUP-PLOG, Turma 3MIEIC03, Grupo Multi-Balance_1

Abstract. Constraint programming aims to solve problems fast than the traditional generate and test method. In this paper we aim to show a way to generate solutions to the Multi Balance Puzzle [2]. We use constraint programming alongside some pre-processing in the SICStus Prolog programming language not only to generate solutions based on existing problems but also to generate new problems.

Keywords: Prolog · Constraint Programming · Fulcrums · Torque · Multi-balance · Puzzle

1 Introduction

With this paper we aim to solve the Multi Balance Puzzle using Constraint Programming. The objective is to provide a fast and optimized solution for a given puzzle and to generate new puzzles of any given size as well. This paper describes the puzzle, alongside its rules and restrictions, our approach to solving and generating new puzzles as well as an analysis of the algorithms developed and the impact of different labeling options.

2 Problem Description

The problem we aim to solve is the Multi Balance Puzzle [2]. It consists of a matrix of arbitrary size, domain and number of fulcrums. A fulcrum is an element that needs to be balanced with equal torques on both sides. In our problem they are represented as black squares or black triangles. The torque equation for each side of a fulcrum is the following:

$$Torque = \sum_{n=1}^{N} (Distance * Force) \tag{1}$$

As can be seen in the following example, the fulcrum is balanced:

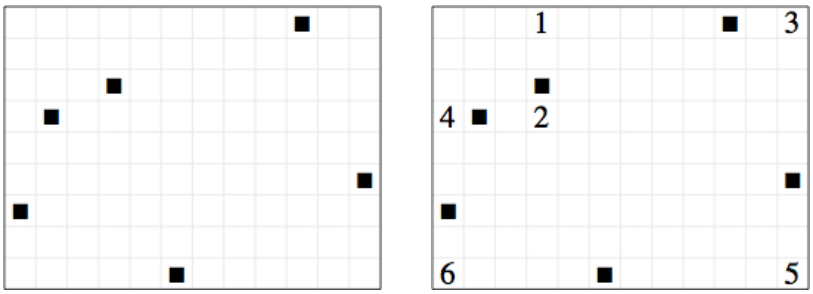
1		2	■	5

$$(3 * 1) + (2 * 0) + (1 * 2) = (1 * 5) \tag{2}$$

We can then surmise our problem as given a matrix with placed fulcrums, how can we distribute all the digits in the domain of 1 to N, where N is a specific integer for each puzzle[1], such that all the following rules are respected:

1. No row or column contains exactly one digit
2. Every row or column that contains 2 or more digits contains exactly one fulcrum
3. Each fulcrum is used horizontally or vertically, but not both
4. Each fulcrum is located where those weights would balance, with equal torques on both sides

An example of solution can be found in the following images:



(a) Puzzle (b) Solution

Fig. 1: Solving Example

3 Approach

Our problem can be modeled as a Constraint Satisfaction Problem (CSP), where the variables are all the cells where a fulcrum is not placed and their domain is 0 to N, with N being the puzzle's domain. The constraints applied are dependent on the position of every fulcrum.

3.1 Pre-processing

Given the very large number of variables, we devised a pre-processing phase to nullify cells based on a new set of rules derived from the puzzle's 4 basic rules:

– If any cell has no fulcrums in its respective column or row, then that cell must be blank

[1] Referenced throughout the paper as the puzzle's Domain

Proof. A line (row or column) with only a digit is impossible, every line with two or more digits must contain exactly one fulcrum. As such, if a cell does not have a fulcrum in both its row and column, then it must not be a digit, otherwise either one of the previous rules would be violated.

– If a fulcrum is touching the left or right border, then that row must not contain a digit

Proof. As a fulcrum can be used either horizontally or vertically but not both, if a Fulcrum is next to the left or right border, the torque will never be balanced, as one of the sides is empty. As such we can conclude that row will have to be exclusively other Fulcrums or blank cells.

– If a fulcrum is touching the upper or lower border, then that column must not contain a digit

Proof. As a fulcrum can be used either horizontally or vertically but not both, if a Fulcrum is next to the upper or lower border, the torque will never be balanced, as one of the sides is empty. As such we can conclude that column will have to be exclusively other Fulcrums or blank cells.

– If there are multiple Fulcrums in a row or column, then said row or column must not contain any digits

Proof. Every line (row or column) cannot contain only one digit. Any line with two or more digits must contain exactly one fulcrum. As such, if a line contains multiple fulcrums, that line must not contain any digits and those fulcrums will be used in the other direction.

The pre-processing phase can be subdivided in 3 steps:

1. Nullify cells whose row or column have no fulcrum
2. If a given fulcrum is touching the border, nullify the respective row or column
3. If a given row or column contains multiple fulcrum, nullify the respective row or column

Some of these rules may overlap, but they all restrict cases the others are unable to pick up. For an example, take the following puzzle (the digits are only included for ease of comparison):

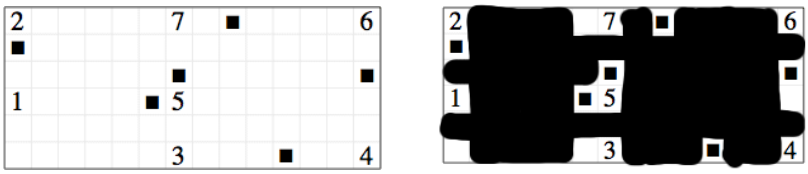
To represent the steps taken by our algorithm applied in order: step 1 in pink, step 2 in blue and step 3 in red.

The impact of the pre-processing stage will be discussed in a later section.

3.2 Decision Variables

The decision variables are every single cell of the matrix that isn't a fulcrum. Every cell has a domain of 0 to N (N being the domain of the puzzle). Thanks to our pre-processing, the total number of variables is decreased drastically, as we determine a vast majority of them to being 0.

A cell with a 0 represents a blank cell, a cell with a -1 represents a fulcrum (they are placed before the domain restriction) and any other number represents the respective digit in the matrix.



(a) Problem (b) Solution

Fig. 2: Pre-processing Example

Fig. 3: Pre-processing Steps

3.3 Constraints

Our solution is exclusively composed of rigid constraints. The predicate responsible for placing our restrictions is restrict_fulcrums\4. For every fulcrum, we get the list of variables at the left, right, up and down directions away from the Fulcrum. Every one of these lists is accompanied by another list stuffed with the distances from the fulcrum, for example, $[1, 2, 3, ..., N]$ for the right list or $[N, ..., 3, 2, 1]$ for the left list.

The scalar product between every pair of lists represents the torque on a given side. As such, we restrict the torque on the right and on the left (H) to be equal and the torque on the upper and lower side (V) to be equal. To ensure rule 3 of our puzzle, we use the following expression to guarantee the fulcrum is only used in one of the directions:

$$(V > 0 \wedge H = 0) \vee (V = 0 \wedge H > 0) \tag{3}$$

The implementation of these constraints using SICStus Prolog are exactly as described above, using built-in predicates such as scalar_product\4.

3.4 Generation

The generation of a new puzzle can is also a CSP, with exactly the same characteristics of the solver, except the entire matrix of variables must be used and their domain now starts at -1 instead of 0, as fulcrums must yet be placed and generated.

Unfortunately, the complexity of generating a puzzle is substantially superior to solving it. We cannot reduce the total number of variables, placing fulcrums greatly increases the number of combinations and

To use one of the puzzles as an example: Given a matrix 8x8, a total of 64 cells, with digits 1 through 7 and 6 fulcrums, we can describe the number of total raw possibilities (ignoring symmetries) to ${}^{64}C_6 \times {}^{58}P_7 = 1.136e^{20}$.

The strategy used in SICStus Prolog was to, for every cell, materialize a variable to indicate if it contained a Fulcrum or not. As there is no way to predict where the labeling will attempt to place the Fulcrums, we must apply a restriction to every single cell using the variable mentioned before and treat it as if it could contain a Fulcrum.

Unlike the restrictions for the solver, we had to deal with extra edge cases that were previously removed by the pre-processing. As such, we had to implement extra boolean conditions to guarantee the correctness of the puzzle generated.

4 Solution Presentation

In this section we'll cover how a solution is displayed.

The predicate responsible for displaying a solution is show_solution/2. It is divided in three parts. First it uses the predicate print_line_top/1, responsible for drawing the top of the matrix. Secondly, the predicate print_board/2, responsible for drawing the middle of the matrix. Finally the predicate print_line_bot/1, responsible for drawing the the bottom of the matrix.

The predicate print_board/2 uses print_line/1 to print each row of the matrix. This predicate is responsible for displaying each element of the row with a separator in the middle of them. For each row, print_board/2 also prints a separator row to format the result as a table.

For each cell, the character printed depends on its type: a fulcrum is represented by a the Unicode character U+9632, a digit is represented by itself and a 0 is represented by a space. Derived from an early version of the display and testing, if a cell is still an unassigned variable, it is represented by a question mark. Every digit also has padding to make it as centered as possible.

The complete representation can be seen in figure 4:

2						7		■					6
■													
						■							■
1					■	5							
						3				■			4

Fig. 4: Display of a solution

5 Experiments and Results

5.1 Dimensional Analysis

To make a dimensional analysis, we tested all the puzzles we had available and plotted the results into a set of charts.

From these charts we can conclude that with our pre-processing technique the time to solve a puzzle remains constant at approximately 0.003 seconds regardless of matrix size, amount of fulcrums or the domain. Now regarding the solutions without the pre-processing we can conclude that what appears to impact the most the solving times are the domain size and the amount of fulcrums. The matrix size does not appear to be a decisive factor in the solving time.

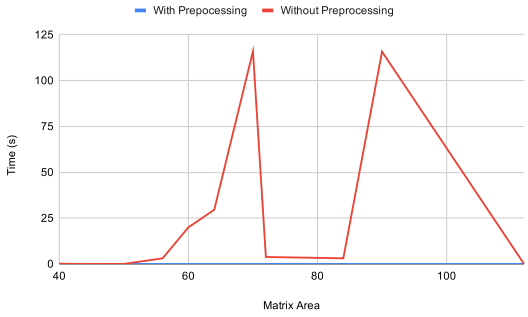
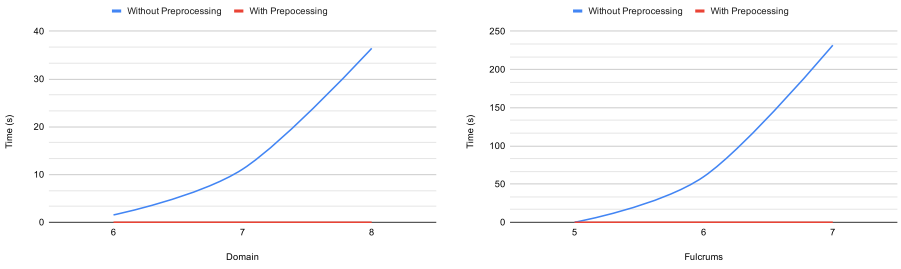


Fig. 5: Time over Matrix Area



(a) Time over Domain

(b) Flower two.

Fig. 6: Time over Fulcrums

5.2 Search Strategies

The SICStus Prolog language allows the choice of different variable and value selection methods which can be consulted here. We applied all of them to 3 of our available puzzles with and without pre-processing. The results indicate that with pre-processing these different methods don´t make the solver faster or slower. On the other hand, without pre-processing, each method had a different effect. To highlight the most impactful ones: ff, ffc and occurrence made a very significant impact reducing the time average of the three puzzles to 0.724s, 0.346s, 1.368s respectively from 33.36s. All other methods altered the time average but not in a very dramatic manner except for max which brought the average up to approximately 175s.

As the pre-processing times are all consistently low, we cannot establish a direct correlation with the best strategies for it. However, we can theorize that the ffc algorithm would be still prove to be the most beneficial one, as it prioritizes the variables with the tightest constraints, therefore reducing the backtracking needed to compute the solution to a minimum.

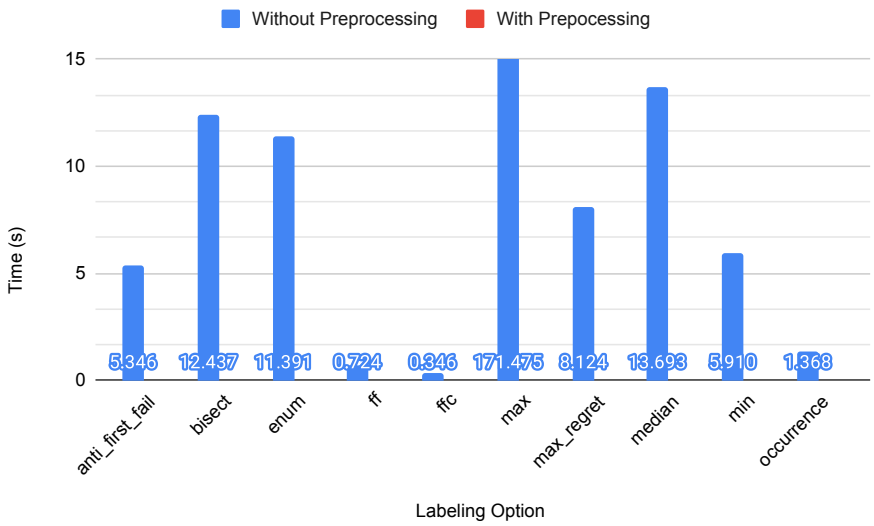


Fig. 7: Display of a solution

5.3 Generation

As discussed previously, the complexity of generating puzzles is much higher than that of solving a puzzle. As such, we have only managed to generate small puzzles, such as 6x8 with a domain of 5 and 5 fulcrums. This took an average

of 33 seconds to find, with ever increasing puzzles not completing in under 15 minutes.

6 Conclusions and Future Work

In this paper we described a way to solve the Multi Balance puzzle using SICStus Prolog and Constraint Logic Programming. We achieved very good results by applying some pre-processing to the puzzle, leaving us with an average solve time of 0.003s on all the puzzles available to us. Therefore we reached the conclusion that this is a very powerful method to tackle this problem. Regarding possible improvements, the solver is already very optimized and any further improvement would not benefit it given the very low solving times. As such, the next big focus would be on improving the generator, as it still does not generate puzzles within an acceptable time frame.

References

1. SICStus Documentation, https://sicstus.sics.se/sicstus/docs/latest4/pdf/sicstus.pdf. Last accessed 4 Jan 2020
2. Puzzle Rules, https://erich-friedman.github.io/puzzle/2Dweight/. Last accessed 4 Jan 2020