Power Strike*

Ângelo Teixeira[up10606516] and Henrique Lima[up10606525]

FEUP-PLOG, 3MIEIC4, *Power Strike_5*

Abstract. The work described in this report consists in the development of a PROLOG program capable of solving problems of the **Power Strike** puzzle using **Constraint Logic Programming (CLP)**. The project development took about ten hours and was developed using mainly a pair programming strategy to guarantee a fluid development. Validation and profiling tests were ran on multiple sample exercises with different labeling options to ensure a functional yet efficient code. Above all, the project shows the impact of labeling options, as well as the overall power of CLP and its application in solving challenges like the one mentioned. By analysing the runtimes of the algorithm for different sample problems, a pattern is visible, however the grouth is close to linear. That analisys can be found at the end of this article.

Keywords: Logic Programming · PLOG · FEUP · Constraint Logic Programming · PROLOG

1 Introduction

This project is part of the goals of the PLOG curricular unit at FEUP and consists of solving the Power Strike puzzle, created by Erich Friedman https://www2.stetson.edu/~efriedma/puzzle/powerstrike/ with the use of CLP in PROLOG. This article is divided in various sections. **Problem Description** explains the problem in its entirety. In the **Approach** section, the methods to solve the problem are explained, in a more technical view, by showing some of the code done and decisions taken. At the **Solution Presentation** section, the solution output is explained and in the **Results** section there are some examples of the program running with some sample problems, and their correspondent performance in terms of runtime, where the different labeling options are compared for each sample problem. Finally, the **Conclusions and Future Work** section contains our review of the work done as well as a perspective on what we could have done further or better.

2 Problem Description

The presented problem consists of solving a numeric puzzle where, in a set of N positive numbers, being $N \geqslant 2$, and given a positive multiplier, each number in

* Logic Programming Course 2018-2019, Power Strike_5

the set is either the result of the previous number multiplied by the multiplier or the result of removing one digit of the previous number, given that it doesn't become zero.

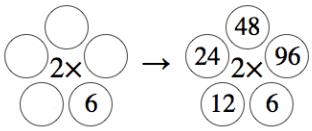

Fig. 1. Visual representation of an example of the puzzle along with a solution. The problem inputs are: $nCells = 5$, $Start = 6$ and $Multiplier = 2$

3 Approach

3.1 Decision Variables

To solve the problem, we used two groups of variables:

1. The numbers in the set (i.e the actual solution numbers)
2. The exponents used, meaning the index of the removed digit when generating numbers by removing a digit from previous ones

That being said, we start with a list on un-initialized variables with the result set size and restrict the first element to be the start number.

Every other number will have a domain of $[1, Max]$, being $Max = Start * Multiplier * (nCells - 1)$ where $nCells$ is the total number of numbers in the problem input.

According to the problem specification, two restrictions are needed:

3.2 Constraints

let MaxAmountOfDigits be the number of digits of the Max possible number in the solution (see previous section)

$$Power \in [1, MaxAmountOfDigits] \tag{1}$$

$$LeftSide = Number \div 10^{Power} \tag{2}$$

$$RightSide = Number \ mod \ 10^{PowerI-1} \tag{3}$$

According to the problem specification, two restrictions are needed to find the next number ($NextNumber$):

1. $NextNumber = PrevNumber * Multiplier$
   (Multiply previous number)

2. $NextNumber = (LeftSide * (PowerI - 1) + RightSide)$
   (Remove one digit from previous number)

To implement these restrictions in Sicstus, the predicates generateNumbers/6 and generateRestrictedNumberRemoveDigit/4 are used, which apply both restrictions to the numbers and generate a restriction when a digit is to be removed.

Sicstus Example 1: The following snippet is part of the generateNumbers/6:

```
generateRestrictedNumberRemoveDigit(
    CurrElement, Coeffs,
    CurrElementWithRemovedDigit, GeneratedPower),

    (NextElement #= CurrElement * Multiplier
        #/\ GeneratedPower #= 1)
    #\/ (CurrElement #>= 10
        #/\ NextElement #= CurrElementWithRemovedDigit)
```

Sicstus Example 2: The following snippet is part of the generateRestrictedNumberRemoveDigit/4:

```
generateRestrictedNumberRemoveDigit(
    element(_, Coeffs, PowerIminus1),
    PowerI #= PowerIminus1 * 10,
    LeftSide #= Number // PowerI,
    RightSide #= Number mod PowerIminus1,
    NextNumber #= (LeftSide * PowerIminus1 + RightSide)
```

In this approach, as it is not possible to make arithmetic operations with non-initialized variables such as $LeftSide = Number \div 10^{Power}$, we keep a list $Coeffs$ that stores decimal coefficients (1, 10, 100, 1000..), holding n values where $n = MaxAmountOfDigits + 1$ which allows for a selection of a value for $Power$, which corresponds to the index of the selected value in $Coeffs$.

3.3 Search Strategy

Upon testing all the labeling options' combinations, it was found that the best combination for a big problem (i.e. a problem with multiple branching points to test) was ff, *bisect* and *up* for the variable selection, branching strategy and value order, respectively. The profiling results for a big problem can be found in Fig.2.

This is a good strategy. Firstly, the variables that have the smallest domain (i.e. least possible solutions) are selected first, which minimizes the backtrack in the future, by minimizing eventual impossible restrictions to these variables. By selecting values for the variables with less options first, it is assured that a solution is found faster, because the rest of the variables have more "room" to adapt

to the propagation of those restrictions. Secondly, *bisect* option is used which accelerates the labeling by partitioning the variable domain in two halves, and propagating that restriction, which narrows the variable domain faster. Lastly, the *up* option makes the labeling more efficient than the opposite *down*. This is due to the fact that when a variable has multiple value options, the smallest one will always be the result of removing a digit from the previous number as those results are smaller than a multiplication. Furthermore, as the multiplier increases, the multiplication will result in a number with more and more digits, which difficults the match at some point with previous numbers in the set. So, if the number of digits is more or less constant, it becomes easier to transform a number back into the original number, making the algorithm run faster.

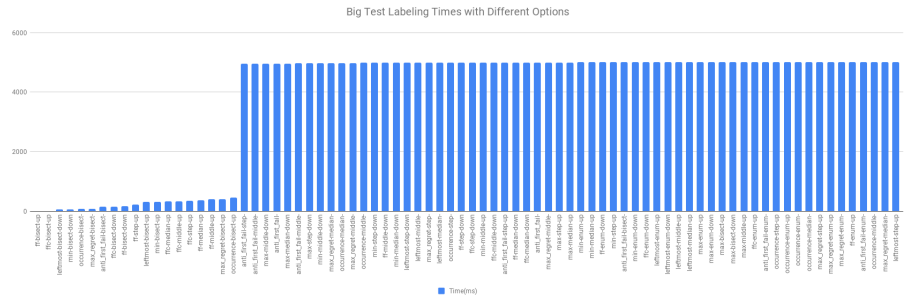

Fig. 2. Runtime of the big example with different combination labeling options.

Further tests can be consulted in the Results section.

4 Solution Presentation

As the solution is internally represented by a List of variables which are then instantiated, the program's output is generated by means of the write/1 predicate, assuming that the last and first elements of the list share the same connection as any other pair of consecutive elements (i.e. the list should be interpreted as a circular list).

5 Results

The figures 3 through 6 are screenshots of the actual PROLOG software running three different samples, each followed by a visual representation of the puzzle and its solution (clockwise).

```
Welcome to power strike!

Please insert the puzzle size (1-3):

1 - Small
2 - Medium
3 - Big
1.

Calculating solution for problem with:
Initial value: 6
Amount of values: 5
Multiplier: 2

The result is: [6,12,24,48,96]
```

Fig. 3. Screenshot of the program output from start to finish - Small Puzzle

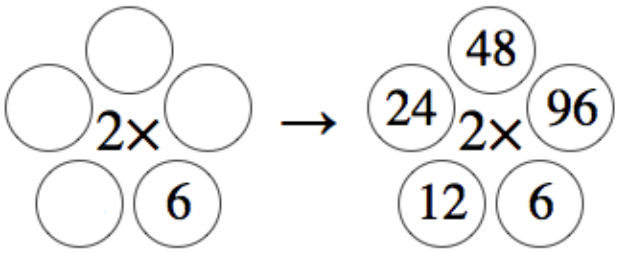

Fig. 4. Visual representation of the Small Puzzle

```
Calculating solution for problem with:
Initial value: 9
Amount of values: 8
Multiplier: 2

The result is: [9,18,36,6,12,24,48,96]
```

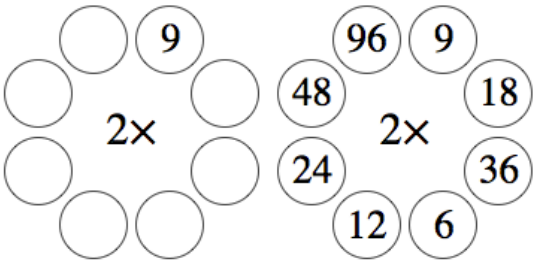

Fig. 5. Screenshot and visual representation of the Medium Puzzle

```
Calculating solution for problem with:
Initial value: 41
Amount of values: 8
Multiplier: 7

The result is: [41,4,28,196,19,9,63,441]
```

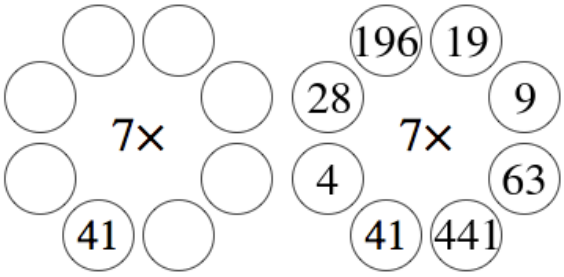

Fig. 6. Screenshot and visual representation of the Big Puzzle

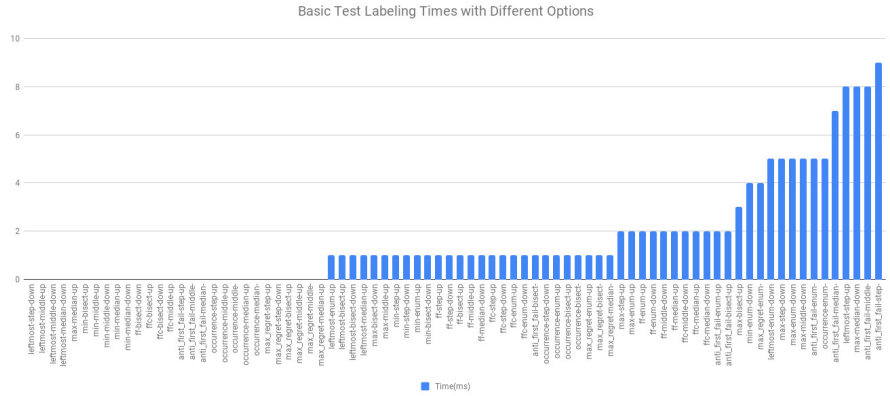

Fig. 7. Run time of the basic example with different combination labeling options.

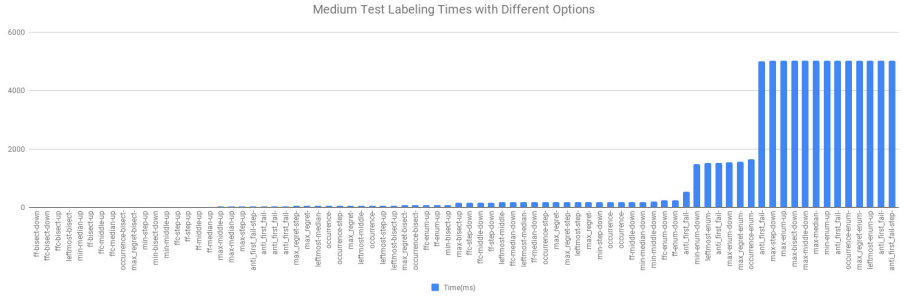

Fig. 8. Run time of the medium example with different combination labeling options.

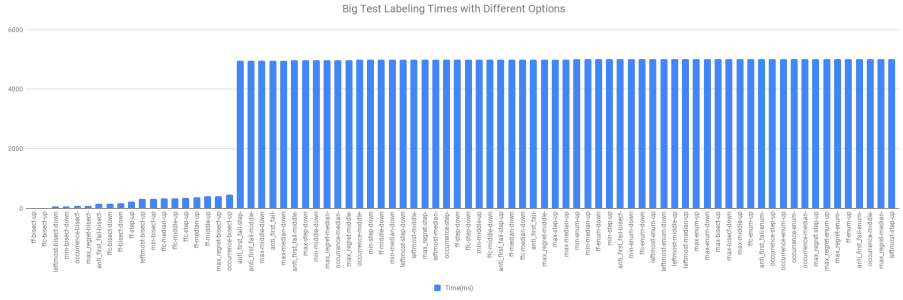

Fig. 9. Run time of the big example with different combination labeling options.

Selecting the labeling options **ff, bisect and up**, further test were executed by varying some input fields and measuring the runtime of the algorithm. The base problem's inputs were $StartNumber = 1$, $Multiplier = 2$ and $nCells = 2$. The tests consisted in the variation of these three variables.

In Figs. 10 to 12, it is possible to see a pattern where every once and a while the runtime is really low, and it grows back up, only to get really low again a few steps after. This is due to the fact that, for some inputs, the start number and the last calculated number can be converted by using one of the two rules of the problem (See **Problem Description** section) without backtracking into other solutions, making the algorithm complete on the first attempts. However, even though it was not possible to test bigger inputs because of integer overflow problems, in Fig. 10 it is possible to see a linear grouth of the run time, for the numbers that require multiple backtracks to get a find a solution.

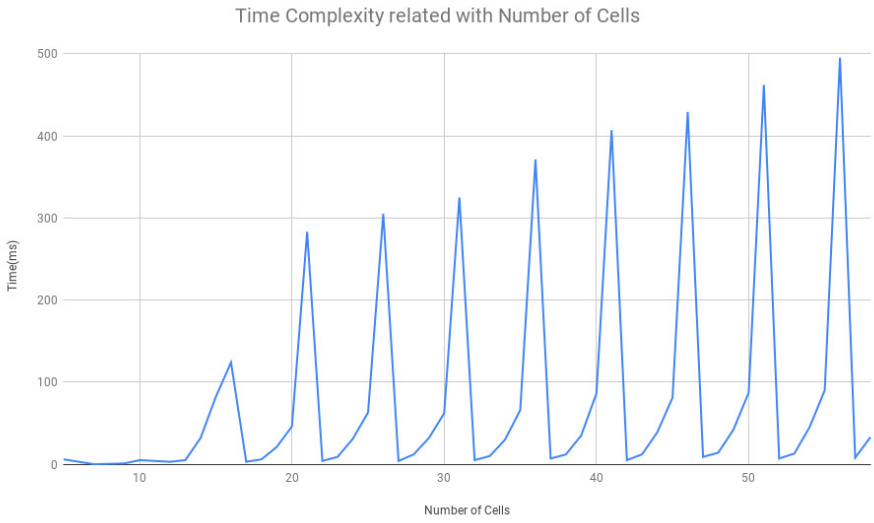

Fig. 10. Run time with different number of cells.

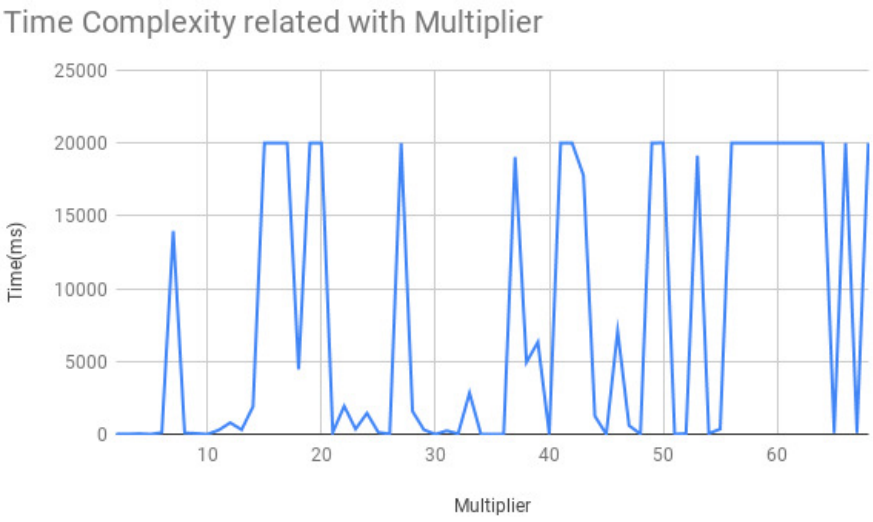

Fig. 11. Run time with different center multipliers.

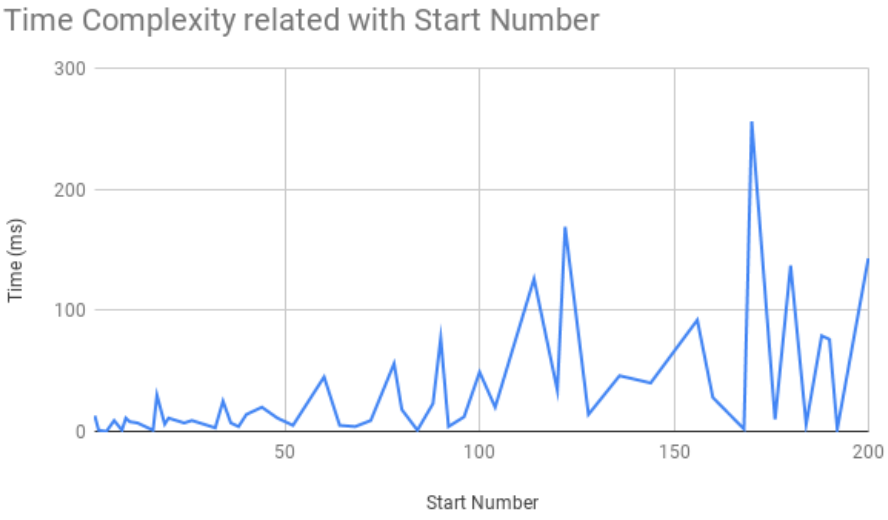

Fig. 12. Run time with different start numbers.

6 Conclusions and Future Work

This project allowed us to work directly with CLP programming in PROLOG and made us aware of its enormous capabilities in terms of puzzle and NP-Complete problem solving. In this concrete project we would like to have done an automaton to apply the removal of a digit restriction, due to its efficiency, when compared to our restrictions, which would imply the creation of an automaton dependent of the number to remove a digit from, where the acceptance states would be the various outputs where a digit from the starting number is removed.

References

1. https://www2.stetson.edu/~efriedma/puzzle/powerstrike/ Erich Friedman (2009)