Resolution and Optimization of a Decision Problem: C-Note

João Sousa and Rafael Ribeiro

FEUP-PLOG, Turma 3MIEIC05, Grupo C-Note_1

Abstract. This project, developed in the context of the curricular unit Logic Programming, using the Prolog environment SICStus, had the objective of solving a decision/optimization problem using restriction programming (SICStus' CLPFD library). The methods used to tackle this problem are described in this article.

Keywords: SICStus · Prolog · C-Note · Restriction programming.

1 Introduction

This project was developed in the context of the final project for the curricular unit Logic Programming, in the 3rd year of the Masters in Informatics and Computing Engineering. The group had to find a possible solution for a decision/optimization problem using restriction programming. From the given options, the group chose a puzzle named C-Note[1], by Erich Friedman, which consists of a square matrix of single digits such that by adding digits before or after these, the sum of all rows and columns add up to 100. The article has the following structure:

- **Problem Description:** detailed description of the problem, including all involved restrictions.
- **Approach:** modeling description of the problem as a Constraint Satisfaction Problem.
  - **Decision Variables:** description of decision variables and respective domains, as well as their meaning in the context of the problem.
  - **Constraints:** description of hard and soft constraints, as well as their implementation using SICStus Prolog.
- **Solution Presentation:** explanation of the predicates that allow the visualization of the solution.
- **Experiments and Results:**
  - **Dimensional Analysis:** examples of execution of the problem with different dimensions and analysis of the obtained results.
  - **Search Strategies:** testing of different search strategies/heuristics, comparing the obtained results.
- **Conclusions and Future Work:** conclusions drawn from the project, advantages and limitations of obtained solution, aspects that can be improved.
- **References:** bibliographic references
- **Attachments:** source code, images and graphs.

2 Problem Description

C-Note is a grid puzzle, much like Sudoku and others of its kind, where the objective is to have all rows and columns sum up to the same integer, in this case, 100. The initial grids consist of single digits and the player must add digits to the left and right of these in order to achieve the previously referred goal.

3 Approach

Since the initial state of the puzzle doesn't require anything beside the values on the cells, the input was represented as a list with the values of the table in sequence

Table 1. Input example

8	8	4
6	2	5
3	6	1

[8, 8, 4, 6, 2, 5, 3, 6, 1]

3.1 Decision Variables

The solution of a puzzle is simply a table with the replaced values, and as such is returned as a list with the same size as the input.

Table 2. Output example

18	8	74
69	26	5
13	66	21

[18, 8, 74, 69, 26, 5, 13, 66, 21]

The domain of these variables are $1..Sum - Size$, since a cell must have a number larger than 0 (e.g if the sum of a row with 3 cells is 100 and no cell can hold 0, the maximum combination is [97, 1, 1]).

3.2 Constraints

The representation of the problem chosen by the group only has 2 hard restrictions:

All rows and columns sum up to 100: Rule imposed by the game. This is assured by the predicates `sum_rows(Rows, Sum)` and `sum_rows(Cols, Sum)`

```
% sum_rows(+Matrix, +S)
sum_rows([], _).
sum_rows([H|T], S):-
    sum(H, #=, S), sum_rows(T, S).
```

All cells in the solution *must include* the digit given in the input: Possible solutions are found through this restriction. Since the game consists of adding digits before or after the initial digit, the final solution will always hold the input value in one of its digits. This is assured by the predicate `includes_digit_list(Input, Output)`

```
% includes_digit(+Var, +Input)
includes_digit(Var, Input):-
    Var mod 10  #= Input.
includes_digit(Var, Input):-
    Var #> 0,
    Rest #= Var // 10,
    includes_digit(Rest, Input).
```

Solution must be unique: This restriction was enforced during the generation of new puzzles, and would drastically increase the time until a valid solution was found. Since the author of the puzzle doesn't specify this as a rule (his first example doesn't obey this), the group decided to leave this feature as optional.

This was achieved with the predicate `more_than_once(Goal)`, [3] which fails if the number of times a Goal predicate succeeds is different from 1. It avoids the program from wasting time on finding all the existing solutions of a puzzle (like it would be done by something like the `findall()` predicate), when it only needs to know if there is one or more possible solutions.

4 Solution Presentation

To visualize this decision problem, the program features a series of menus to lead the user to all the functionalities. To startup the program, after consulting the file, simply run the predicate `cnote`.

```
  /$$$$$$                      /$$   /$$  /$$$$$$  /$$$$$$$$ /$$$$$$$$
 /$$__  $$                    | $$$ | $$ /$$__  $$|__  $$__/| $$_____/
| $$  \__/                    | $$$$| $$| $$  \ $$   | $$   | $$
| $$             /$$$$$$      | $$ $$ $$| $$  | $$   | $$   | $$$$$
| $$            |______/      | $$  $$$$| $$  | $$   | $$   | $$__/
| $$    $$                    | $$\  $$$| $$  | $$   | $$   | $$
|  $$$$$$/                    | $$ \  $$|  $$$$$$/   | $$   | $$$$$$$$
 \______/                     |__/  \__/ \______/    |__/   |________/


1. Solve input puzzle
2. Generate puzzle
3. Generate puzzle with unique solution
0. Exit

:-
```

One can input an existing configuration through **option 1** (check examples here). Through options **2 and 3**, the program will generate and then, if the user wishes, solve a new random puzzle. The latter restricts the generated puzzle so it has an **unique** solution.

5 Experiments and Results

To test the chosen approach, the group ran tests varying both the dimension of the problem and the search heuristic used in labeling.

5.1 Dimensional Analysis

The first variable to be tested was the **size of the puzzle**, where the values ranging from 3*3 to 7*7 were used. For this, puzzles with target sum 60 were generated for each test. Default labeling options were used. Graphs for this table found in the attachments. 1 2 3

From these we extrapolate that the increases in time and backtracks are exponential while the increase in constraints seems rather erratic. This may be explained by the randomness of the generated inputs.

The other variable that was tested was the target sum of the rows and columns. Puzzles were also generated for these runs, with a size of 3*3, and default labeling options were also used. Similarly, graphs can be found in the attachments. 4 5 6

Table 3. Varying Input Size

	Time	Backtracks	Constraints created
100	0.00	202	152
150	0.02	1753	597
200	0.03	2665	877
250	0.04	2627	845
300	0.14	7265	1741
400	1.00	43848	4294
500	0.79	26961	1725
700	1.28	42719	2142
900	2.03	512087	2119
1200	47.04	2971885	202
1500	33.78	1099845	7124

Table 4. Varying Target Sum

	Time	Backtracks	Constraints created
3	0	70	93
4	0.01	2068	1481
5	0.18	35773	9164
6	6	12447174	7296845
7	116.5	20342573	581

5.2 Search Strategies

Tests were also done in the hopes of finding the most efficient search strategy used in the labeling process. Below is the table of runtimes for all combinations of arguments of the predicate `labeling([], Output)`[2]. The options `leftmost, smallest, largest, first_fail, anti_first_fail, occurrence, most_constrained and max_regret` control the order in which the next variable is selected for assignment and the options `step, enum, bisect, middle and median` determine the way in which choices are made for the selected variable. These tests were all run on the 4*4 puzzle `[5, 2, 7, 1, 7, 1, 5, 9, 8, 8, 3, 7, 5, 2, 9, 1]`, with a target sum of 100.

Table 5. Search strategies runtimes

	leftmost	smallest	largest	first_fail	anti_first_fail	occurrence	most_constrained	max_regret
step	0.4	1.78	2.54	4.44	0.71	0.39	3.85	0.4
enum	0.39	0.42	3.5	6.11	0.93	0.51	6.24	0.52
bisect	0.32	0.84	5.26	3.17	1.76	0.33	3.09	0.32
middle	0.58	0.52	4.28	7.06	33.86	0.62	7.05	0.62
median	0.59	0.53	4.28	6.97	45.53	0.62	7.03	0.62

From this we conclude the most efficient combination of options is **bisect & max_regret**, and that the option **anti_first_fail** yielded the worst results.

6 Conclusions and Future Work

The project's goal of applying the knowledge taught in practical and theoretical PLOG classes was achieved, showing how useful restriction programming can be in some cases.

Some of the bigger obstacles faced were "switching" from the common imperative programming paradigm to the declarative style used, as well as the implementation of some of the restrictions. The predicate `includes_digit(Var, Input)` was the most challenging to crack, and was a big breakthrough in our understanding of the CLPFD library.

Our solution could be improved upon, especially when it comes to generating unique solutions, which proved to be the slowest part of the program. We are not sure if much else in terms of restrictions and labeling could be done under the rules of the existing game.

To conclude, the project was a success in contributing positvely to our knowledge of Prolog and restriction programming, having definetely piqued our interest in this way of problem solving.

References

1. Erich Friedman C-Note official page `https://erich-friedman.github.io/puzzle/100/`
2. SICStus Documentation v.latest4 `https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/Enumeration-Predicates.html`
3. Ulrich Neumerkel, Institut für Information Systems Engineering, Wien Austria Europe `http://www.complang.tuwien.ac.at/ulrich/sicstus-prolog/call_nth.pl`
4. Useful command to clear the console `https://swi-prolog.discourse.group/t/useful-command-to-clear-the-console/976`

7 Attachments

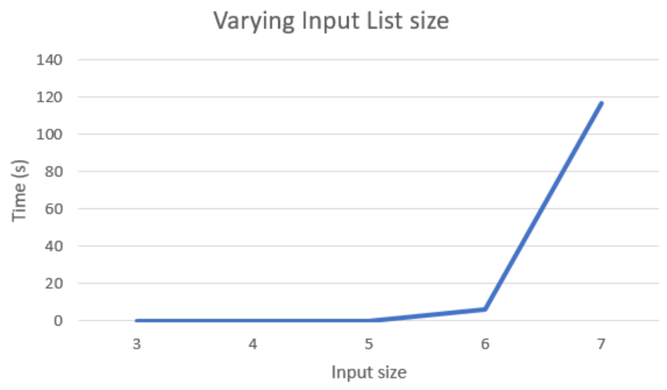

Fig. 1. Graph for varying input size and time

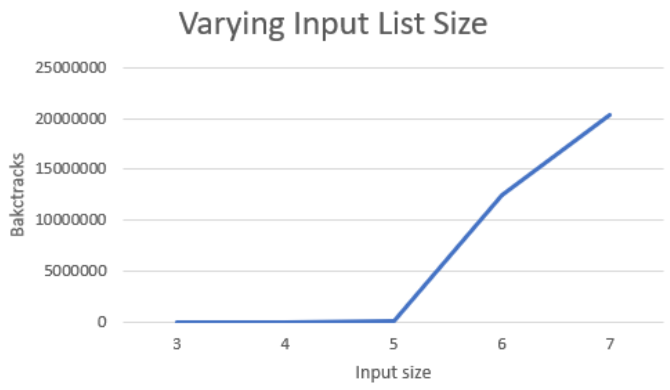

Fig. 2. Graph for varying input size and number of backtracks

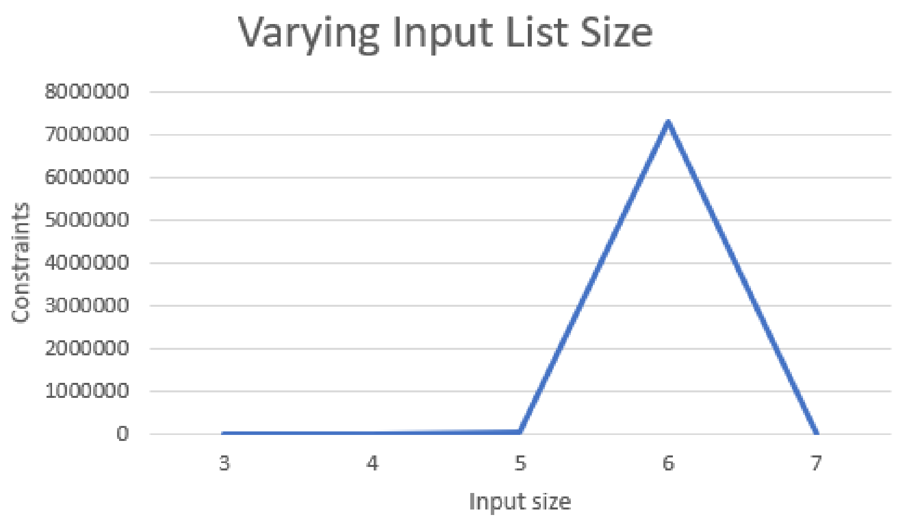

Fig. 3. Graph for varying input size and number of constraints

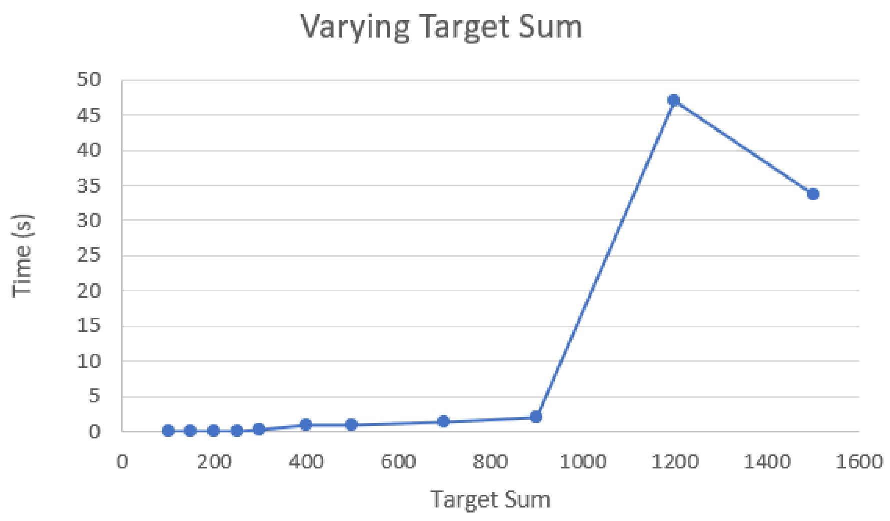

Fig. 4. Graph for varying target sum and time

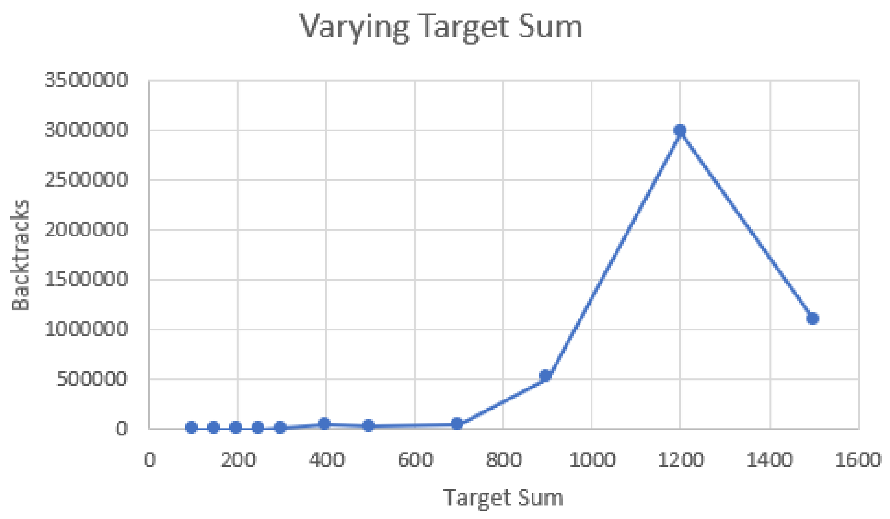

Fig. 5. Graph for varying target sum and number of backtracks

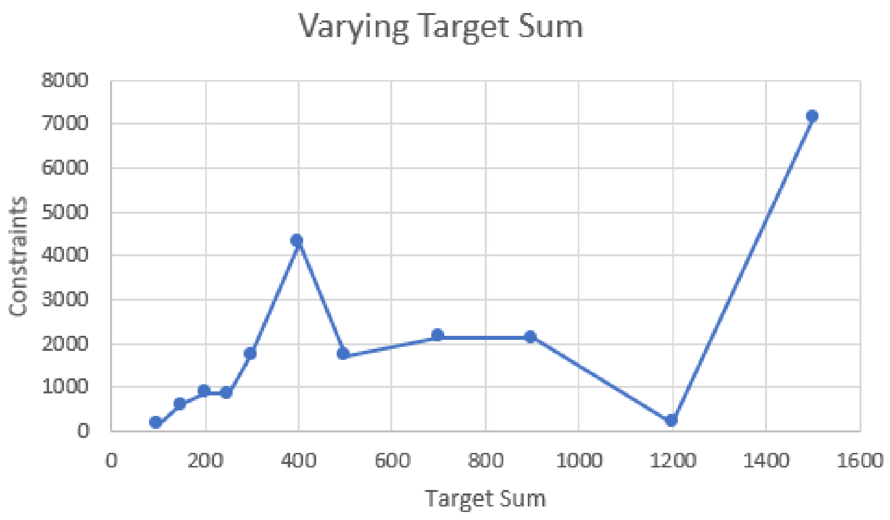

Fig. 6. Graph for varying target sum and number of constraints