Master in Informatics and Computing Engineering

PLR

Report TP

Valentina Wu - 201907483
Víctor Saldanha Nunes - 201907226

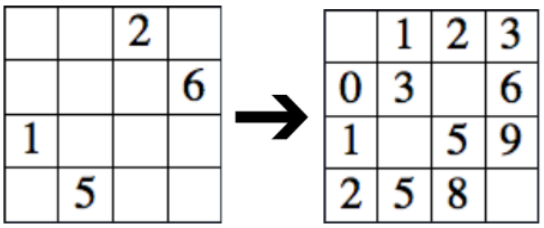
FEUP - MEIC, 22/23

1 - Introduction

Logic programming with constraints is a powerful paradigm for solving complex optimization problems. It combines the expressiveness of logic programming with the ability to impose constraints on the solutions, leading to efficient and elegant solutions.
In this discipline, we will approach a problem Arithmetic Sequence Puzzles to solve in logic programming with constraints, implementing in two different platforms: prolog and CPlex, and draw conclusions based on the comparison with that implementation.

2 - Problem Definition

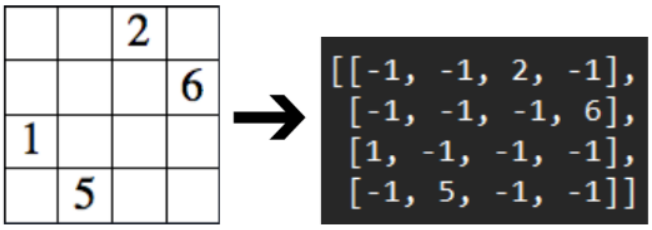
The problem consists of the following puzzle: It receives a matrix, containing one number in each row and column. The challenge is to make each row and column have exactly three distinct numbers from 0 to 9, that follows an ascending order (from left to right for the rows and from top to down for the columns) and have the same increment between the three numbers (the increments isn't necessarily the same for all of the rows or columns).
In the following image, we can see an example of a initial matrix and it's solution:



So, as we can see, the initial number on the row and column is not always the first, the last or the middle one. This randomness will add more complexity to the problem solution and will make different search strategies result in different times to find the solution. Even for initial matrices with the same dimensions, the time taken for the same search strategy to find a solution can vary a lot.

3 - Problem Modelation

The problem was modelated in the following way. The matrix is represented as a list of lists, and the inner lists represent each row. The element -1 represents an empty square, whereas any other number represents a square filled with that number.



4 - The Prolog Solution

4.1 - Algorithm Explanation

First, let's start by explaining the logic behind the prolog solution.

4.1.1 - Solution Overview

The main predicate is **solve/2**, which receives an input matrix of the initial problem and returns the output matrix which is the it's solution.

```
%solve(+InputBoard, -OutputBoard)
solve(InputBoard, OutputBoard) :- statistics(walltime, [Start,_]),
                                  length(InputBoard, BoardSize), length(OutputBoard, BoardSize),
                                  length(PositionsHBoard, BoardSize), length(PositionsVBoard, BoardSize),
                                  checkHorizontal(InputBoard, OutputBoard, PositionsHBoard),
                                  checkVertical(InputBoard, OutputBoard, PositionsVBoard),
                                  append(PositionsHBoard, PlainPositionsHBoard),
                                  append(PositionsVBoard, PlainPositionsVBoard),
                                  append(PlainPositionsVBoard, PlainPositionsHBoard, Vars),
                                  labeling([dom_w_deg, median], Vars),
                                  statistics(walltime, [End,_]), Time is End - Start,
                                  format('Time spent to find the answer: ~3d s~n', [Time]).
```

It starts making the input and output board have the same size. The predicates **checkHorizontal/3** and **checkVertical/3** will apply the appropriate constraints to respectively each row and each column.
The variables **PositionsHBoard** and **PositionsVBoard** are a list of lists that contains the three numbers of the row/column and their position. These are the variables that will be searched in the labeling phase.

4.1.2 - Solving each Row and Column

The predicate checkHorizontal/3 calls **solveLine/3** for each individual row to constrain it. The checkVertical/3 simply transpose the board, turning the columns into rows, and calls checkHorizontal/3 to solve it.

```
%checkHorizontal(+InputBoard, -OutputBoard)
checkHorizontal([],[],[]).
checkHorizontal([InputLine | InputRemainingLines], [OutputLine | OutputRemainingLines],
                [PositionsLine | PositionsRemainingLine]) :- solveLine(InputLine, OutputLine, PositionsLine),
                                                             checkHorizontal(InputRemainingLines, OutputRemainingLines, PositionsRemainingLine).

%solveLine(+InputLine, -OutputLine)
solveLine(InputLine, OutputLine, [N1, N1Position, N2, N2Position, N3, N3Position]) :- length(InputLine, LineSize), length(OutputLine, LineSize),
                                  findLineNumber(InputLine, N3, N3Position),
                                  element(N3Position, OutputLine, N3),
                                  domain(OutputLine, -1, 9),
                                  domain([N1, N2], 0, 9), all_distinct([N1, N2, N3]),
                                  domain([N1Position, N2Position], 1, LineSize), all_distinct([N1Position, N2Position, N3Position]),
                                  constrainNumbers([N1, N2, N3], [N1Position, N2Position, N3Position]),
                                  element(N1Position, OutputLine, N1), element(N2Position, OutputLine, N2),
                                  NumberOfEmptySquares is LineSize - 3,
                                  exactly(-1, OutputLine, NumberOfEmptySquares).
```

```
checkVertical(InputBoard, OutputBoard, PositionsVBoard) :- transpose(InputBoard, InputBoardTransposed), transpose(OutputBoard, OutputBoardTransposed),
                                                           checkHorizontal(InputBoardTransposed, OutputBoardTransposed, PositionsVBoard).
```

```
%findLineNumber(+InputLine, -N3, -N3Position)
findLineNumber(InputLine, N3, N3Position) :- element(N3Position, InputLine, N3), N3 #\= -1.
```

First, using **findLineNumber/3** the algorithm finds out the number and its position from the input row/column, which will be N3 and N3Position. After that, it does intuitive constraints about the N1, N2, N3 and their position.
Then, the **constrainNumbers/2** will do the major constraints with the N1, N2, N3 and their position. This predicate will be explained in details on the next subsection.

4.1.3 - Constraining the numbers

```
%constrainNumbers(+[N1, N2, N3], +[N1Position, N2Position, N3Position])
constrainNumbers([N1, N2, N3], [N1Position, N2Position, N3Position]) :- C in 1..4,
                                                                        (N3 #= N2 + C #/\ N2 #= N1 + C #/\ N1Position #< N2Position #/\ N2Position #< N3Position) +
                                                                        (N2 #= N3 + C #/\ N3 #= N1 + C #/\ N1Position #< N3Position #/\ N3Position #< N2Position) +
                                                                        (N3 #= N1 + C #/\ N1 #= N2 + C #/\ N2Position #< N1Position #/\ N1Position #< N3Position) +
                                                                        (N1 #= N3 + C #/\ N3 #= N2 + C #/\ N2Position #< N3Position #/\ N3Position #< N1Position) +
                                                                        (N2 #= N1 + C #/\ N1 #= N3 + C #/\ N3Position #< N1Position #/\ N1Position #< N2Position) +
                                                                        (N1 #= N2 + C #/\ N2 #= N3 + C #/\ N3Position #< N2Position #/\ N2Position #< N1Position) #= 1.
```

The **constrainNumbers/2** will apply the order constrain to the three numbers and their position. The C variable will force the numbers to have a fixed increment between them.
As we don't know the order of the numbers, we must write all the 6 scenarios of the order of N1, N2 and N3.

4.2 - Statistics

In order to make the statistics, several labeling options were tested, with different variable ordering and selection.
For one specific board, we found the following results:

Input matrix:

			0			
	1					
					6	
				8		
						7
		6				
7						

The results can be found in the prolog section, in the annex, test all the combinations of order and selection variable.

As we can see, the best ones were the ffc, dom_w_def and ff. If we only looked in the specific example, we would think the ffc method is clearly the best, but we noticed that the times taken to solve the problem can vary a lot even between puzzles with same dimension. That's why we made a more general table, with more results, and the mean results where the following:

Variable Ordering (time/s)

Method	4x4	5x5	6x6	7x7
leftmost	0,014s	1,310s	48,957s	X
min	0,066s	63,881s	X	X
max	2,001s	X	X	X
ff	0,012s	0,614s	1,208s	294,055s
anti_first_fail	5,536s	X	X	X
occurrence	0,015s	11,993s	23,983s	X
ffc	0,014s	0,296s	3,087s	12,881s
max_regret	0,005s	1,542s	10,243s	X
impact	0,004s	9,522s	X	X
dom_w_deg	**0,013s**	**0,056s**	**0,644s**	**7,083s**

Testing the variable selection of the ffc and the dom_w_deg, we found the following results:

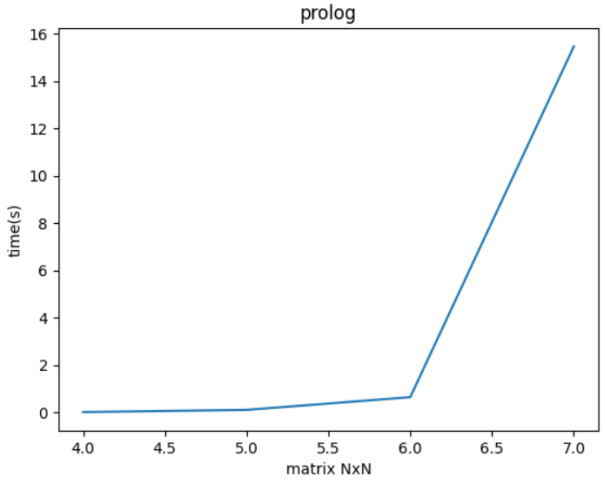
Variable Selection (time/s)

Method	6x6	7x7
ffc step	0,579s	20,104s
ffc enum	0,568s	19,784s
ffc bisect	0,578s	20,096s
ffc median	0,134s	89,664s
ffc middle	0,267s	58,175s

Variable Selection (time/s)

Method	6x6	7x7
dom_w_deg step	0,182s	19,011s
dom_w_deg enum	0,552s	42,620s
dom_w_deg bisect	0,164s	17,115s
dom_w_deg median	**0,644s**	**15,452s**
dom_w_deg middle	5,308s	98,907s

Then, we can conclude that, in a general way, **dom_w_deg median** is the best combination to use, even though it is not the best one in some particular cases.



5 - The CPlex Solution

5.1 - Algorithm Explanation

In this section, we will explain the logic behind the CPlex solution.

5.1.1 - CPlex Variables

The variable we defined is shown in the code

```
using CP;

int n = ...; //Matrix nxn
int InputBoard[1..n][1..n] = ...;

range Rows = 1..n;
range Cols = 1..n;
range Numbers = 0..9;

dvar int OutputBoard[1..n][1..n] in -1..9;
dvar int N1_Rows[Rows] in Numbers;
dvar int N2_Rows[Rows] in Numbers;
dvar int N3_Rows[Rows] in Numbers;
dvar int N1_Cols[Cols] in Numbers;
dvar int N2_Cols[Cols] in Numbers;
dvar int N3_Cols[Cols] in Numbers;
dvar int N1Position_Rows[Rows] in Cols;
dvar int N2Position_Rows[Rows] in Cols;
dvar int N3Position_Rows[Rows] in Cols;
dvar int N1Position_Cols[Cols] in Rows;
dvar int N2Position_Cols[Cols] in Rows;
dvar int N3Position_Cols[Cols] in Rows;
```

n is the size of the matrix, *InputBoard* is the matrix that we received with initial values and defines some ranges necessary for the program.
Then we define several decision variables, *OutputBoard* as the name indicate is the result of executing the program, then the rest decision variable is the three number of each row and column, as well as, their positions respectively.

5.1.2 - CPlex Constraints

In this subsection, we will explain the constraints defined in our program:

The first one is to allocate the values in the correct place, we find all the numbers different of -1 in the InputBoard and put it as on the value N3 (similar to *findLineNumber* defined in prolog), then with the values and positions represented as decision variable we match the three values in each row and column of the OutputBoard (the same thing as the function element in prolog).

```
38  forall (r in Rows, c in Cols){
39    //findLineNumber(InputLine, N3, N3Position),
40   (InputBoard[r][c] != -1) => (N3_Rows[r]==InputBoard[r][c] && N3Position_Rows[r] == c) &&
41               (N3_Cols[c]==InputBoard[r][c] && N3Position_Cols[c] == r);
42
43   /* element(N1Position, OutputLine, N1)
44   element(N2Position, OutputLine, N2)
45   element(N3Position, OutputLine, N3)*/
46   N1_Rows[r] == OutputBoard[r][N1Position_Rows[r]];
47   N2_Rows[r] == OutputBoard[r][N2Position_Rows[r]];
48   N3_Rows[r] == OutputBoard[r][N3Position_Rows[r]];
49
50   N1_Cols[c] == OutputBoard[N1Position_Cols[c]][c];
51   N2_Cols[c] == OutputBoard[N2Position_Cols[c]][c];
52   N3_Cols[c] == OutputBoard[N3Position_Cols[c]][c];
53  }
```

The next constraint is the main rule of the problem, we need to know the ordering of the three values either in the row or in the column, so we can determine the sequence of positions of the numbers and what the number adding to the smallest ones gives the next value, looking on the first condition N1<N2<N3 we can imply that N2-N1 is equal to N3-N2 and the ordering of positions is the same of the numbers.

```
64  //constrainNumbers(+[N1, N2, N3], +[N1Position, N2Position, N3Position])
65  forall(j in Cols){
66    // Rows
67    (N1_Rows[j] < N2_Rows[j] && N2_Rows[j] < N3_Rows[j]) => N2_Rows[j]-N1_Rows[j]==N3_Rows[j]-N2_Rows[j] &&
68       N1Position_Rows[j] < N2Position_Rows[j] && N2Position_Rows[j] < N3Position_Rows[j];
69    (N1_Rows[j] < N3_Rows[j] && N3_Rows[j] < N2_Rows[j]) => N3_Rows[j]-N1_Rows[j]==N2_Rows[j]-N3_Rows[j] &&
70       N1Position_Rows[j] < N3Position_Rows[j] && N3Position_Rows[j] < N2Position_Rows[j];
71    (N2_Rows[j] < N1_Rows[j] && N1_Rows[j] < N3_Rows[j]) => N1_Rows[j]-N2_Rows[j]==N3_Rows[j]-N1_Rows[j] &&
72       N2Position_Rows[j] < N1Position_Rows[j] && N1Position_Rows[j] < N3Position_Rows[j];
73    (N2_Rows[j] < N3_Rows[j] && N3_Rows[j] < N1_Rows[j]) => N3_Rows[j]-N2_Rows[j]==N1_Rows[j]-N3_Rows[j] &&
74       N2Position_Rows[j] < N3Position_Rows[j] && N3Position_Rows[j] < N1Position_Rows[j];
75    (N3_Rows[j] < N1_Rows[j] && N1_Rows[j] < N2_Rows[j]) => N1_Rows[j]-N3_Rows[j]==N2_Rows[j]-N1_Rows[j] &&
76       N3Position_Rows[j] < N1Position_Rows[j] && N1Position_Rows[j] < N2Position_Rows[j];
77    (N3_Rows[j] < N2_Rows[j] && N2_Rows[j] < N1_Rows[j]) => N2_Rows[j]-N3_Rows[j]==N1_Rows[j]-N2_Rows[j] &&
78       N3Position_Rows[j] < N2Position_Rows[j] && N2Position_Rows[j] < N1Position_Rows[j];
79
80    //Cols
81    (N1_Cols[j] < N2_Cols[j] && N2_Cols[j] < N3_Cols[j]) => N2_Cols[j]-N1_Cols[j]==N3_Cols[j]-N2_Cols[j] &&
82       N1Position_Cols[j] < N2Position_Cols[j] && N2Position_Cols[j] < N3Position_Cols[j];
83    (N1_Cols[j] < N3_Cols[j] && N3_Cols[j] < N2_Cols[j]) => N3_Cols[j]-N1_Cols[j]==N2_Cols[j]-N3_Cols[j] &&
84       N1Position_Cols[j] < N3Position_Cols[j] && N3Position_Cols[j] < N2Position_Cols[j];
85    (N2_Cols[j] < N1_Cols[j] && N1_Cols[j] < N3_Cols[j]) => N1_Cols[j]-N2_Cols[j]==N3_Cols[j]-N1_Cols[j] &&
86       N2Position_Cols[j] < N1Position_Cols[j] && N1Position_Cols[j] < N3Position_Cols[j];
87    (N2_Cols[j] < N3_Cols[j] && N3_Cols[j] < N1_Cols[j]) => N3_Cols[j]-N2_Cols[j]==N1_Cols[j]-N3_Cols[j] &&
88       N2Position_Cols[j] < N3Position_Cols[j] && N3Position_Cols[j] < N1Position_Cols[j];
89    (N3_Cols[j] < N1_Cols[j] && N1_Cols[j] < N2_Cols[j]) => N1_Cols[j]-N3_Cols[j]==N2_Cols[j]-N1_Cols[j] &&
90       N3Position_Cols[j] < N1Position_Cols[j] && N1Position_Cols[j] < N2Position_Cols[j];
91    (N3_Cols[j] < N2_Cols[j] && N2_Cols[j] < N1_Cols[j]) => N2_Cols[j]-N3_Cols[j]==N1_Cols[j]-N2_Cols[j] &&
92       N3Position_Cols[j] < N2Position_Cols[j] && N2Position_Cols[j] < N1Position_Cols[j];
93  }
```

We need to imply that three numbers in the same row or column must be different also the position needs to be different to guarantee we have three numbers in each row and column, so the following constraint is produced.

```
55  /*all_distinct([N1, N2, N3])
56   all_distinct([N1Position, N2Position, N3Position])*/
57  forall(i in Rows){
58    N1_Rows[i] != N2_Rows[i] && N2_Rows[i] != N3_Rows[i] && N1_Rows[i] != N3_Rows[i];
59    N1_Cols[i] != N2_Cols[i] && N2_Cols[i] != N3_Cols[i] && N1_Cols[i] != N3_Cols[i];
60    N1Position_Rows[i] != N2Position_Rows[i] && N2Position_Rows[i] != N3Position_Rows[i] && N1Position_Rows[i] != N3Position_Rows[i];
61    N1Position_Cols[i] != N2Position_Cols[i] && N2Position_Cols[i] != N3Position_Cols[i] && N1Position_Cols[i] != N3Position_Cols[i];
62  }
```

The final constraint is filling the squares, basically filling the rest of the values with positions different than positions N1, N2, and N3 as -1.

```
forall(r in Rows, c in Cols){

    (N1Position_Rows[r] == c) => OutputBoard[r][c] == N1_Rows[r];
    (N2Position_Rows[r] == c) => OutputBoard[r][c] == N2_Rows[r];
    (N3Position_Rows[r] == c) => OutputBoard[r][c] == N3_Rows[r];
    (N1Position_Rows[r] != c && N2Position_Rows[r] != c && N3Position_Rows[r] != c) => OutputBoard[r][c] == -1;

    (N1Position_Cols[c] == r) => OutputBoard[r][c] == N1_Cols[c];
    (N2Position_Cols[c] == r) => OutputBoard[r][c] == N2_Cols[c];
    (N3Position_Cols[c] == r) => OutputBoard[r][c] == N3_Cols[c];
    (N1Position_Cols[c] != r && N2Position_Cols[c] != r && N3Position_Cols[c] != r) => OutputBoard[r][c] == -1;

 }
```

5.1.3 - CPlex Settings

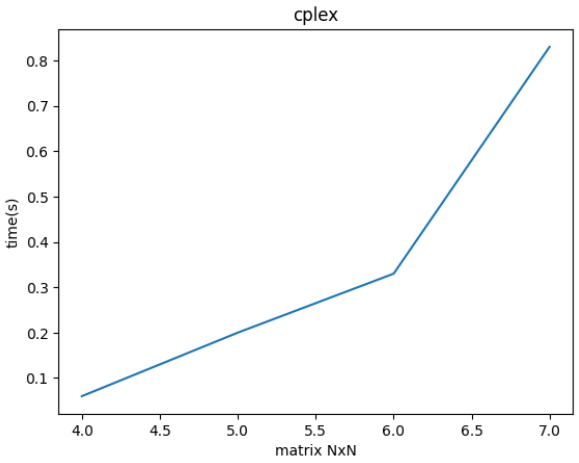
The setting we use is described as follows:

```
execute{
  var f = cp.factory;
  var phase1 = f.searchPhase(OutputBoard,
       f.selectSmallest(f.successRate()),f.selectSmallest(f.valueImpact()));
  cp.setSearchPhases(phase1);
  cp.param.SearchType = "MultiPoint";
}
```

To obtain the settings described above, we experiment with multiple combinations: the 4 search types (Depth-first, Restart, Multi-point, Iterative-diving) and variable and value selection (selectSmallest(eval), selectLargest(eval), selectRandomVar()); then we choose the option with the shorter time. The results are presented in tables in the Annex (SEARCH_TYPE__Depth_first, SEARCH_TYPE__Restart, SEARCH_TYPE__Multi_point, SEARCH_TYPE__Iterative_diving).

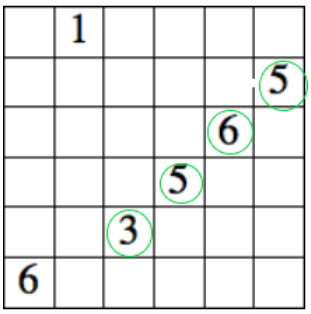
5.2 - CPlex Experimental Statistics

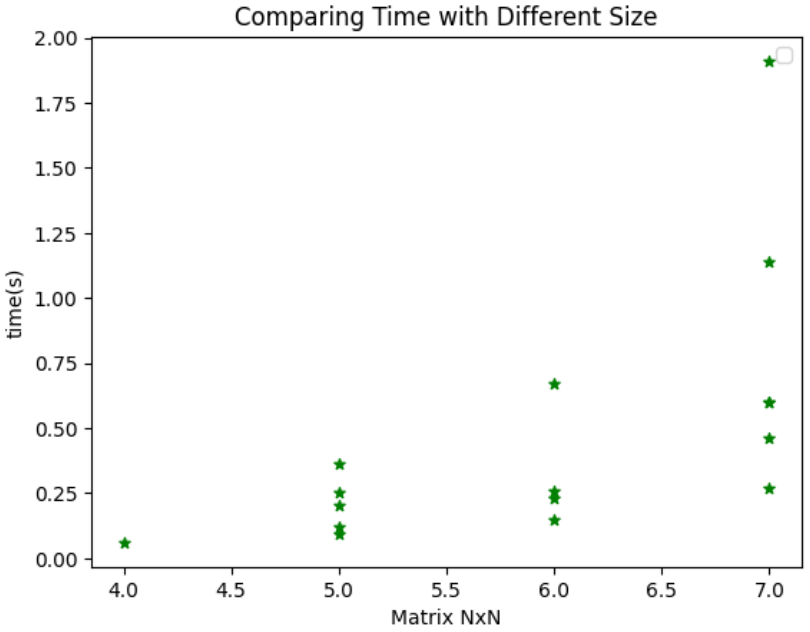
The average of the time obtained to use the best setting is shown in the graph below. The time required to solve a puzzle can grow exponentially, which didn't happen in this cplex solution due to the fact that the potential performance of CPlex.

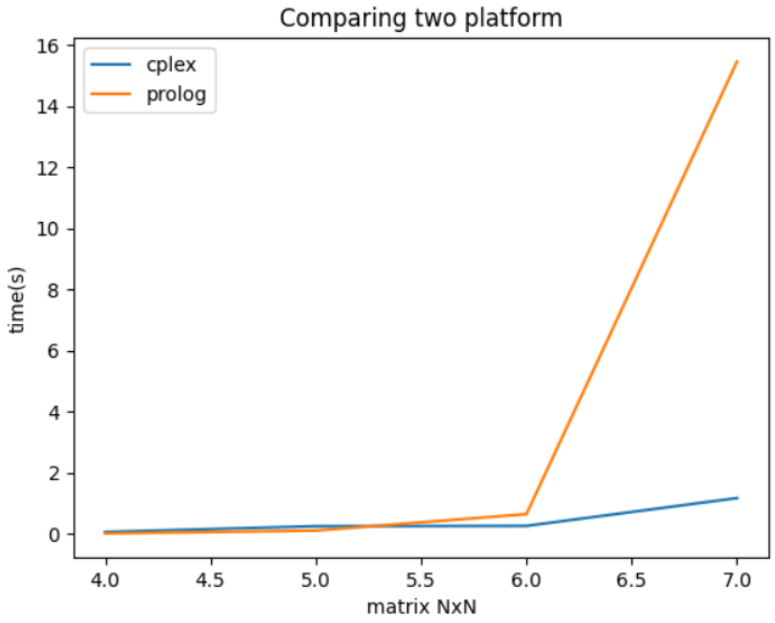


6 - Results

We noticed an interesting thing, the time in a matrix of the same size differs widely, so we tried to look for similarities in those who spent less time and we found that initial values of the matrix with different order affect the result. We conclude possibly a matrix with a diagonal right filled with some initial values gives a better result.







The aside graph shows the growth of the time with two different platforms, as we can see the performance of the CPlex is better than Prolog.

7 - Conclusion and Future Work

In conclusion, the results were really satisfactory, especially the CPlex program.The difference of time taken to solve the problem between them can be explained by the different algorithm implementation, because each language has a different way of approach and write a solution which is readable and intuitive to the programer, using the language tools. Even though the CPlex solution is better in general, it's worth noting that the prolog approach is better in general for small dimension puzzles.
For future work, it would be a good idea to explore and understand with more examples and different settings why the time with the matrix of the same size presents some variation and we can categorize to resolve the puzzle with the best settings for that case.

Annex

https://docs.google.com/spreadsheets/d/1MMGoeGotxOmSP0sj_d2rQlWfM8GRAvyP/edit?usp=sharing&ouid=103699860397206286644&rtpof=true&sd=true

References

https://erich-friedman.github.io/puzzle/arith/