Crypto-Product Puzzle

Diogo Rosário and Henrique Ribeiro

FEUP-PLOG

Turma 3MIEIC03
Grupo Crypto-Product_1

Abstract - This project had the aim of understanding better how to work with constraints in SICStus Prolog. In order to achieve this, we had to choose between a variety of puzzles so we ended up choosing crypto-product. Our project can solve and generate puzzles of several sizes due to the constraints set by us. In the end we concluded that constraints are very helpful and effective at solving these types of problems

1 Introduction

This project was developed using the SICStus Prolog Development Environment during the 3rd year course Logic Programming, part of the Integrated Master's Degree in Informatics and Computing Engineering at FEUP. The main goal of this project was to build a program using Constraint Logic Programming capable of solving and generating Crypto-Product puzzles developed by Erich Friedman.
To execute the program in order to solve pre-loaded puzzles or randomly generate new ones use the predicate *crypto_product/0.* This will take you to menu where you can decide which you want. If you want to have a problem solved simply use the predicate *crypto_user/3* which receives three lists corresponding to three numbers (for example, crypto_user([R],[G,R],[B,G]) will solve the puzzle shown below).
The article has the following structure:

- Problem Description: Detailed puzzle description and rules
- Approach: Approach used to solve these puzzles using constraints
  - Decision Variables: domains and variable description
  - Constraints: constraints implemented and description
- Solution Presentation: Description of predicates used to view solution in a human friendly way
- Results: Analysis of different sized problems

  - Dimensional Analysis: Analysis of time spent solving puzzles of different sizes
  - Search strategies: Analysis of different heuristics

- Conclusions and future work: conclusions from this project
- References: Materials consulted to develop this project
- Annex: source code, results

2 Problem Description

A Crypto-product puzzle consist of an equation of different colors, each color corresponding to a number. In *Fig. 1* we can see an unsolved Crypto-product puzzle.

Fig. 1. Example of an unsolved puzzle

To solve this puzzle each color must be assigned a number, so that the result is equal to the product of the other 2 numbers. In *Fig. 2* we can see the solution to the puzzle in *Fig. 1*.

Fig. 2. Example of a solved puzzle

Each color must have its value unique, that is, there cannot be 2 different colors with the same number and that number must be a single digit, between 0 – 9.

3 Approach

Since our goal was to have a program capable of both solving and generating puzzles, we had to make two different approaches. Both were approached with *Constraint Logic Programming*.

3.1 Solver

Decision Variables.

For our solver we decided to use three lists, each representing a part of the multiplication (the two numbers and the result).

```
puzzle(1,[R],[G,R],[B,G]).
```

Using this template, we can easily apply constraints to obtain a correct solution to our problem. With the help of *setValDomain/4* we are able to set the domain of every number represented by a letter within the in order to have correct numbers to work with.

Constraints.

As mentioned before, our puzzles have two important constraints:

- Each different color must have a different number associated.
- The same color has the same number associated.
- The product must be correct.

For the first two constraints we had to append all input lists into one to be able to start applying the correct restrictions.

Once we had that done, we needed to remove all duplicate values from it using *remove_dups/2*, because if we had any duplicates the next step would not work. Lastly, we used *all_distinct/1* to force every different variable to have a different corresponding value.

The last constraint needed a bit more thought to it. The first step was creating a valid number from the input lists given. This was obtained by iterating all the lists in order and for each variable in the list we would add it to an accumulator, but before we did that we would multiply the accumulator by ten in order to shift the number to the left.

After that we would need to limit the values each variable could have. Since the first digit of each number can't be 0, the first time we called the auxiliar predicate we would set the domain to be between 1 and 9, the next recursive calls would set the domain to be between 1 and 9.

To facilitate this, we decided to create an auxiliar function *setValDomain/4* which would receive the list of colors and apply the constraints.

```
setValDomain([],_,IntValue, IntValue).
setValDomain([X|Rest], L, IntValue, Acc):-
    X in L..9,
    NextAcc #= Acc*10 + X,
    setValDomain(Rest, 0, IntValue, NextAcc).
```

3.2 Generator

To make the generator we had to think the opposite of when we were making the solver. Instead of having a list with colors and turning them into a number, we started with a number and turned it into a list of colors.

Decision Variables.

For the generator the only decision variable are the two numbers in the multiplication. The domain for these numbers depends on the difficulty of the desired puzzle.

For the difficulty 1 the domain of the numbers is between 1 and 10, for difficulty 2 the domain is between 10 and 100 and for difficulty 3 we decided to make a custom generator which the user inputs a custom domain for each side which ranges from 1000 to 999999999.

Constraints.

For the generator we only needed to have a different number of constraints for the different difficulties.

- The domain of the numbers is a constraint present in all difficulties.
- The number of different variables is present in difficulties 1 and 2.

Although the domain of the numbers is present in all difficulties, it is obtained by different methods.

For difficulties 1 and 2 we simply calculate the power of 10 to the difficulty and the power of 10 to the difficulty-1 and apply those two values as the upper and lower bound.

For the third difficulty we have a completely different method.

We start by asking the user for the number of digits he would like the two number of the product to have. After the user has input two numbers (both between 4 and 9), we calculate two auxiliary numbers which are the number of digits minus 1 for both numbers. This way we can set the lower bound of the domain to be the power of ten to the auxiliary value calculated and the upper bound to be the power of ten to the number of digits minus 1 (for example if the number of digits is 4, the lower bound will be 10^3 = 1000 and the upper bound is 10^4 – 1 = 9999).

The second constraint is only used in difficulties 1 and 2 and is obtained by using an auxiliary predicate to make sure we use the right output.

```
make_distinct(Difficulty, Output):-
    (Difficulty #= 1 #<=> Output).

generatePuzzle(Difficulty):-
  …
  (Output #= 1) #=> (K #= 3 #\/ K #= 2),

  (Ouput #= 0) #=> (K #= 6 #\/ K #= 5 #\/ K #= 4 #\/ K #= 3
), nvalue(K,F),
  …
```

This makes it so the generated puzzle is not only different colors (for example, it can't be B x G = PR).

On top of this, we also force the number to have a fixed length, for example, for the difficulty 1 the problem must have a total of 4 digits.

Randomizing the puzzle.

To randomize the generated puzzle, we used the options value and variable when labeling.

The value predicate is the following:

```
mySelValores(Var, _Rest, BB, BB1) :-
    fd_set(Var, Set), fdset_to_list(Set, List),
    random_member(Value, List),
    (
        first_bound(BB, BB1), Var #= Value
        ;
        later_bound(BB, BB1), Var #\= Value
    ).
```

The variable predicate is the following:

```
selRandom(ListOfVars, Var, Rest):-
    random_select(Var, ListOfVars, Rest).
```

These two options combined we can guarantee a randomization of the problem generated.

In order to change the value of each color for different puzzles (for example, to prevent the color green to always have the value 1), we used the predicate *random_permutation/2* to have a list of colors randomly ordered.

4 Solution presentation

To present the solution, we simply used the predicate *print_results/3* which received the three lists and converted them into an equation of type X * Y = Z.

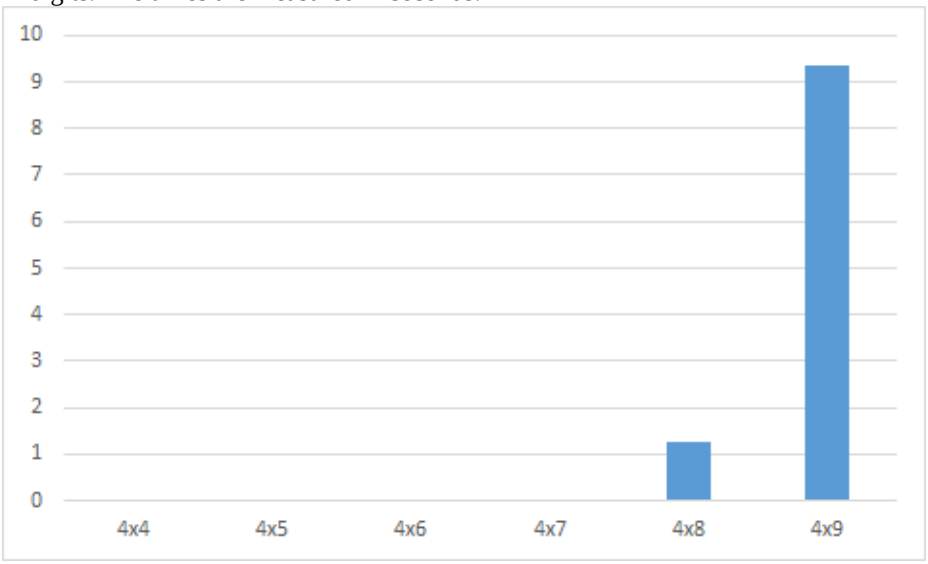
This predicate made use of *printList/1* which prints a list recursively.

5 Results

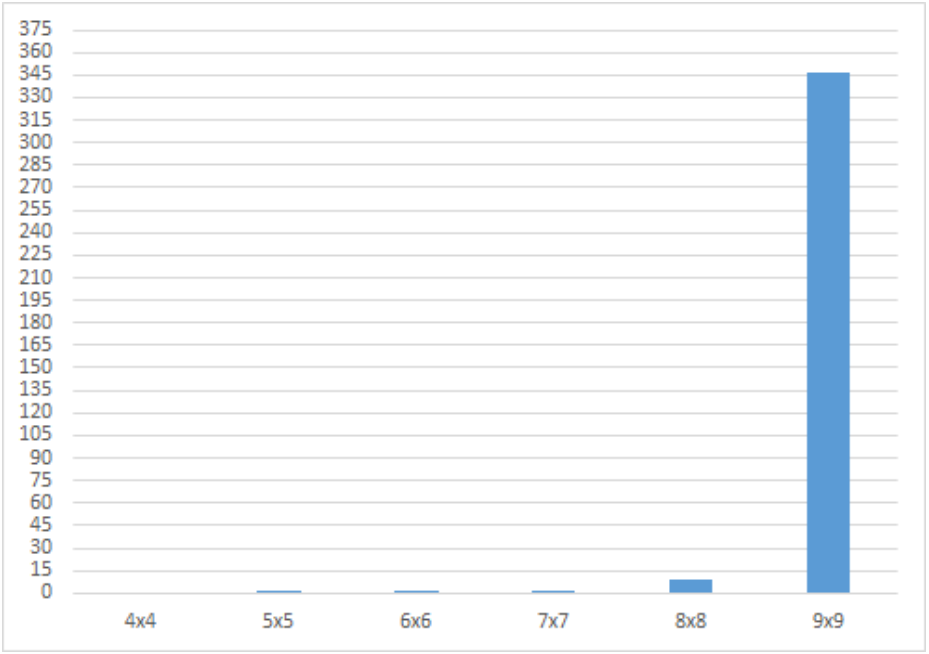
5.1 Dimensional Analysis

To test the dimensional analysis, we decided to use the generator due to the fact that increasing the dimension of the problem the solver had to solve would only increase the time by some milliseconds which weren't relevant. In the next 2 images we can

see that the time spent generating problems increases exponentially with the increase in digits. The times are measured in seconds.



Time spent to generate a puzzle multiplying numbers with N digits * M digits

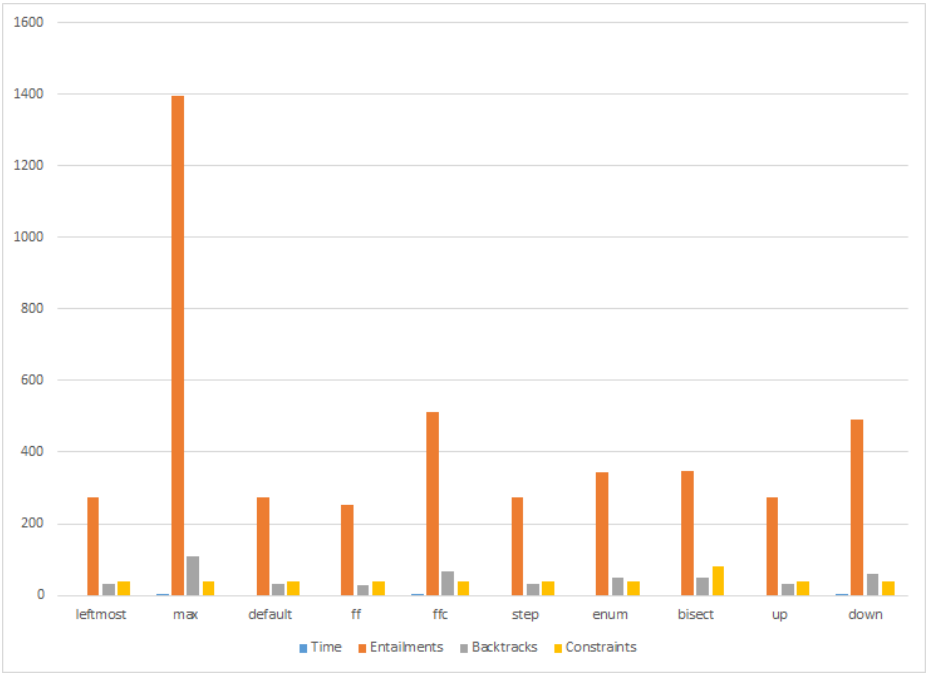


Time spent to generate a puzzle multiplying numbers with N digits * N digits

5.2 Search Strategies

In the next 2 graphs we used the solver using a 9x9 puzzle (BCPTBCGOG X MPTPBPYWP = COPMWGPOCPOYOCRPC). The times were measured in milli-seconds.
The results of fd_statistics are visible, separated in 2 graphs due to the huge difference of values between Resumptions/Prunings and the other values. We can conclude that to solve a crypto-product puzzle the worst search strategies are max, down, ffc.

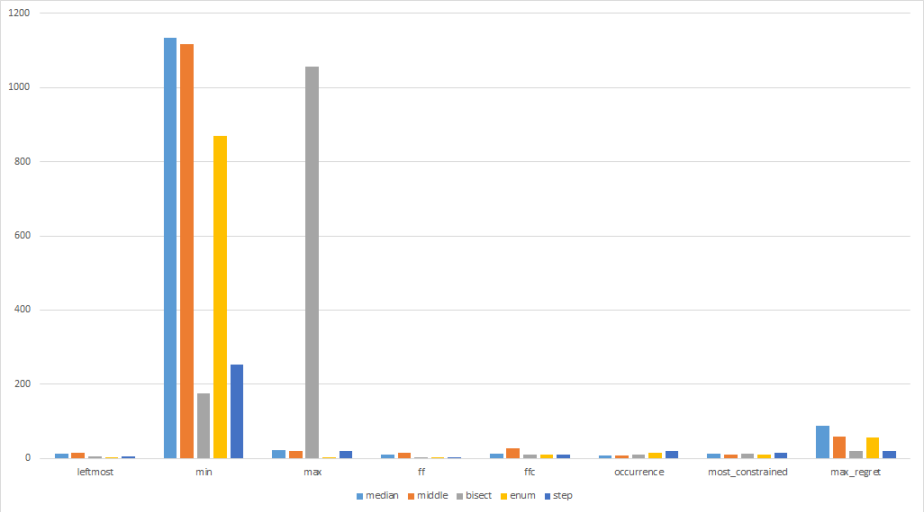
Resumptions and prunings on a 9x9 puzzle



Time, entailments, backtracks and constraints on a 9x9 puzzle

In the following graph we can see the times in milliseconds using different search options combined. We can conclude that min is also one of the worst strategies using almost any other value search. There is also a case where max has very bad results, which is when it's combined with bisect.

It is also noticeable that the best results, although very similar, are with ff variable search, combined with enum.



6 Conclusion

In conclusion, with this puzzle solving problem the group learned how to use constraints towards a simple goal and its computing power.

During the development of the project, the group discovered the importance of constraint programming as some constraints (such as setting the domain of variables to test every possible solution) would be very difficult to make when working without them or very inefficient.

We also learned that different methods to obtain the variable and the value can have a very big impact on the performance of the program.

Our solution has a big advantage: it's very fast and easy to use. Getting your problems solved with our program is very simple and fast, which is very important.

Although the solver is fast, the generator is a bit slow for big numbers which, with some more study about restrictions and random generation, could be fixed.

7 Annex

cryptoProduct.pl

```
:- use_module(library(lists)).
:- use_module(library(between)).
:- use_module(library(clpfd)).
:- use_module(library(random)).
```

```
:- include('solver.pl').
:- include('menus.pl').
:- include('puzzles.pl').
:- include('difficultiesToValues.pl').

crypto_product :-
    prompt(_,''),
    main_menu(Mode),
    mode_menu(Mode).
```

puzzles.pl

```
% pre-loaded puzzle to test the solver.
puzzle(1,[R],[G,R],[B,G]).
puzzle(2,[B],[B,G],[R,R,R]).
puzzle(3,[G],[G,B],[B,R,G]).
puzzle(4,[R],[B,R],[B,G,B]).
puzzle(5,[B],[R,G],[R,R,G]).
puzzle(6,[G,R],[R,G],[R,B,R]).
puzzle(7,[R],[G,B],[B,G,G]).
puzzle(8,[B,R],[R,G],[G,B,G]).
puzzle(9,[G,B],[G,R],[R,B,B]).
puzzle(10,[R,B],[B,B],[G,R,G]).
puzzle(11,[B,G],[B,R],[G,B,R]).
puzzle(12,[G],[G,R],[R,G,G]).
puzzle(13,[R],[R,B],[G,B,G]).
puzzle(14,[B],[B,R],[G,R,R]).
puzzle(15,[G],[G,B],[B,B,R]).

% available colors for the puzzle generator
colors(['R','G','B','W','O','Y','P','C','M','T']).
```

difficultiesToValues.pl

```
% number of digits for the generator's difficulties
getDiffLength(1,4).
getDiffLength(2,8).
```

menus.pl

```
mode_menu(1):-
    repeat,
```

```
        nl, write('   ------------------------------------
-------------------------------------------------------
'), nl, nl,
        write('1) R x GR = BG'), nl,
        write('2) B x BG = RRR'), nl,
        write('3) G x GB = BRG'), nl,
        write('4) R x BR = BGB'), nl,
        write('5) B x RG = RRG'), nl,
        write('6) GR x RG = RBR'), nl,
        write('7) R x GB = BGG'), nl,
        write('8) BR x RG = GBG'), nl,
        write('9) GB x GR = RBB'), nl,
        write('10) RB x BB = GRG'), nl,
        write('11) BG x BR = GBR'), nl,
        write('12) G x GR = RGG'), nl,
        write('13) R x RB = GBG'), nl,
        write('14) B x BR = GRR'), nl,
        write('15) G x GB = BBR'), nl,
        catch(read(Puzzle),_,true), number(Puzzle),
        between(1,15,Puzzle), !,

    puzzle(Puzzle,L1,L2,Sol),
    crypto_user(L1, L2, Sol).

mode_menu(2):-
    repeat,
        nl, write('   ------------------------------------
-------------------------------------------------------
'), nl, nl,
        write('1) Easy .'), nl,
        write('2) Medium.'), nl,
        write('3) Custom.'), nl,
        catch(read(AuxDiff),_,true), number(AuxDiff),
        between(1,3,AuxDiff), !,
    Difficulty = AuxDiff,
    generatePuzzle(Difficulty).
```

```
main_menu(Mode):-
    write('Welcome to crypto products!'),
     repeat,
        nl, write('   ------------------------------------
------------------------------------------------------
'), nl, nl,
        write('1) Solve a puzzle.'), nl,
        write('2) Generate a random puzzle.'), nl,
        catch(read(AuxMode),_,true), number(AuxMode),
        between(1,2,AuxMode), !,
    Mode = AuxMode.
```

solver.pl

```
% used to limit the values each variable on a list can have
 (lim-
its them to be between 1 and 9 if the number is the first o
ne or 0 and 9 if not)
% turns the whole list into one number (for example the lis
t [X,Y,Z] is redifined as the number XYZ)
setValDomain([],_,IntValue, IntValue).
setValDomain([X|Rest], L, IntValue, Acc):-
    X in L..9,
    NextAcc #= Acc*10 + X,
    setValDomain(Rest, 0, IntValue, NextAcc).

% prints a list
printList([]).
printList([X|R]):-
    write(X),
    printList(R).

% for-
mat-
ed way to print the results of either the solver or the ran
dom generator
print_results(MultLeft,MultRight,Sol):-
    printList(MultLeft),
    write(' x '),
    printList(MultRight),
```

```
    write(' = '),
    printList(Sol).

% reset_timer :- statistics(walltime,_).
% print_time :-
%   statistics(walltime,[_,T]),
%   TS is ((T//10)*10)/1000,
%   nl, write('Time: '), write(T), write('ms'), nl, nl.

% solver for the crypto puzzle
crypto_user(MultLeft, MultRight, Sol):-
    setValDomain(MultLeft,1, LeftVal, 0),
    setValDomain(MultRight,1, RightVal, 0),
    setValDomain(Sol,1, SolVal, 0),

    append(MultLeft,MultRight, L),
    append(L, Sol, F),
    remove_dups(F,NoDups),
    all_distinct(NoDups),
    SolVal #= LeftVal * RightVal,

    % reset_timer,
    labeling([ ff, enum ],NoDups),
    % print_time,
    % fd_statistics,

    print_results(MultLeft,MultRight,Sol).
```

generator.pl

```
% con-
verts a number to a list with it's digits (for example 123
becomes [1,2,3])
convertNumberToList(0,List,List).
convertNumberToList(Number,List,Acc):-
    Number #> 0,
    NewNumber #= Number div 10,
    Digit #= Number mod 10,
```

```
    NewAcc = [Digit|Acc],
    convertNumberToList(NewNumber,List,NewAcc).

% used to randomize the labeling result
mySelValores(Var, _Rest, BB, BB1) :-
    fd_set(Var, Set), fdset_to_list(Set, List),
    random_member(Value, List),
    (
        first_bound(BB, BB1), Var #= Value
        ;
        later_bound(BB, BB1), Var #\= Value
    ).

% selects a random variable
selRandom(ListOfVars, Var, Rest):-
    random_select(Var, ListOfVars, Rest).

% makes sure some numbers are repeated on the puzzle (for e
xam-
ple turns G x R = BP in valid since no color is repeated)
% the number of repeated variables depends on the difficult
y of the puzzle
make_distinct(Difficulty, Output):-
    (Difficulty #= 1 #<=> Output).

% codi-
fies a number into a list of colors (for example 121 become
s ['G','R','G']) using a list with a randized order of colo
rs
% so that in different puzzles the same number is represent
ed by a diferent color
getCharsList([],_,[]).
getCharsList([HN|TN],Colors,[Elem|ET]):-
    nth0(HN,Colors,Elem),
    getCharsList(TN,Colors,ET).

generatePuzzle(Difficulty):-
    AuxDiff is Difficulty - 1,
```

```
    MaxNumberLength is integer(exp(10,Difficulty)),
    MinNumberLength is integer(exp(10, AuxDiff)),
    X in MinNumberLength..MaxNumberLength,
    Y in MinNumberLength..MaxNumberLength,
    X #> Y,
    X #\= Y,
    Result #= X * Y,

    convertNumberToList(X,L1,[]),
    convertNumberToList(Y,L2,[]),
    convertNumberToList(Result,Sol,[]),
    append(L1,L2, L),
    append(L, Sol, F),

    make_distinct(Difficulty, Output),

    (Output #= 1) #=> (K #= 3 #\/ K #= 2),

    (Out-
put #= 0) #=> (K #= 6 #\/ K #= 5 #\/ K #= 4 #\/ K #= 3),
    nvalue(K,F),

    length(F,Leng),
    getDiffLength(Difficulty,ExpectLeng),
    Leng#=ExpectLeng,

    label-
ing([value(mySelValores), variable(selRandom)],[X,Y]),

    colors(Aux), % gets a list with 10 colors
    ran-
dom_permutation(Aux,Colors), % randimizes the list of color
s so that in different puzzles the same number is represent
ed by a diferent color
    getCharsList(L1,Colors, LChar1),
    getCharsList(L2,Colors, LChar2),
    getCharsList(Sol,Colors, SolChar),
```

```
    print_results(LChar1,LChar2,SolChar). % displays the ra
ndom puzzle



generatePuzzle(3):-
 % used to obtain the number of digits each number will hav
e in a custom puzzle
    repeat,
        write('Number of digits for one number (between 4 a
nd 9)'), nl,
        catch(read(DigitX),_,true), number(DigitX),
        between(4,9,DigitX), !,
    repeat,
        write('Number of digits for the other number (betwe
en 4 and 9)'), nl,
        catch(read(DigitY),_,true), number(DigitY),
        between(4,9,DigitY), !,
    generateCustomPuzzle(DigitX, DigitY).


generateCustomPuzzle(DigitX, DigitY):-
    % use to limit the number of digits for X (for example,
 if the user wants 4 digist the domain becomes 1000 to 9999
)
    AuxX is DigitX - 1,
    LowerX is integer(exp(10,AuxX)),
    UpperX is integer(exp(10,DigitX)) - 1,

    % use to limit the number of digits for Y (for example,
 if the user wants 4 digist the domain becomes 1000 to 9999
)
    AuxY is DigitY - 1,
    LowerY is integer(exp(10,AuxY)),
    UpperY is integer(exp(10,DigitY)) - 1,

    X in LowerX..UpperX,
    Y in LowerY..UpperY,
```

```
    Result #= X * Y,

    convertNumberToList(X,L1,[]),
    convertNumberToList(Y,L2,[]),
    convertNumberToList(Result,Sol,[]),

    % reset_timer,
    label-
ing([value(mySelValores), variable(selRandom)],[X,Y]),
    % print_time,
    % fd_statistics,

    colors(Aux),
    random_permutation(Aux,Colors),
    getCharsList(L1,Colors, LChar1),
    getCharsList(L2,Colors, LChar2),
    getCharsList(Sol,Colors, SolChar),
    print_results(LChar1,LChar2,SolChar).
```