The use of restrictions in Logic Programming: Puzzle Akkoy

Filipa Ramos and Ines Santos

Faculdade de Engenharia da Universidade do Porto,
Rua Dr. Roberto Frias, 4200 - 465, Porto, Portugal
{up201305378,up201303501}@fe.up.pt
FEUP-PLOG,Turma3MIEIC06,GrupoAkkoy_2
https://sigarra.up.pt/feup/pt/web_page.inicial

Abstract. The present report serves the purpose of explaning the process of finding a solution for akkoy puzzles using programming with restrictions. It also refers the implementation of restrictions in order to generate random puzzles. The main objective is to deepen the knowledge of *PROLOG*, specially the clpfd library. The project was developed for the curricular unit of Logic Programming. The results will be evaluated in order to realize the efficiency of the found solution.

Keywords: restrictions programming logic prolog efficiency

1 Introduction

This project was developed for the curricular unit of Logic Programming in order to deepen the knowledge of the clpfd prolog library. Between the many objectives accounted for this project, the following can be highlighted: examining the results of the use of restrictions whilst programming, understanding the logic of rule-based languages, realizing the advantages of logic in programming. The analized puzzle has many restrictions which made it hard to find a solution which incorporated all the restrictions. Besides this, the restrictions must be applied to different objects such as columns, rows and areas. The implemented solution uses the following approach: find out the different possibilities for each column and row according to the numbers given by the problem. The board that fills the restrictions in every row and column will be the solution. The returned board is a list of variables. Each number one represents a black square and each number zero represents a white square. The article is structured in order to make it easier to understand the solution. Firstly, the problem is described. Secondly, the solution and its visualization is explained. Finally, the results are analized and the conclusions are drawn.

2 Problem Description

The presented problem consists in solving an akkoy puzzle. This puzzle has a blank board with numbers on the top and on the right. Figure 1 represents a puzzle with size seven.

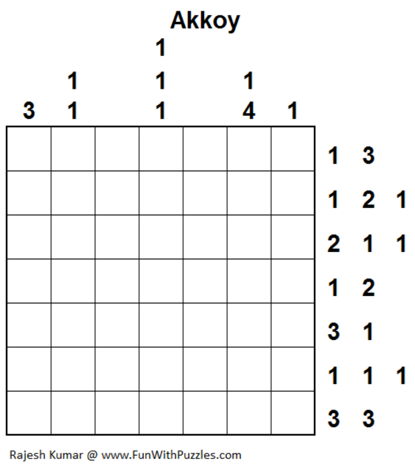


Fig. 1. Example Puzzle.

The numbers on the top represent the number of black squares in each column. For example, in the fourth column there must be three black squares separated. The numbers on the right represent the number of white squares in each line. If these requirements are met, the solution will be a drawing composed of black and white areas (such as Figure 2).

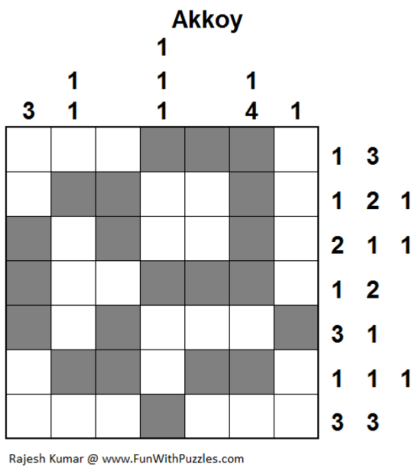


Fig. 2. Solution of the example puzzle.

If there are no numbers in a certain line or column it means that there are no restrictions on the respective line or column (restriction represented by [x]).

The puzzles created dynamically in the implementation consisted of assigning random colors to an empty board.

3 Approach

3.1 Decision Variables

The following are the several decision variables used in the developed project:

1. `getPossibilities(S, Begins, R1)`
   (a) S represents the size of the row
   (b) Begins are the decision variables
   (c) R1 is the list with the restriction for the column
   (d) domain is between 1 and S

   - Returns a list with the possible index positions for areas in a row according to the given restrictions. The domain is from one to S because it represents all the possible positions on the row.
2. `apply_restrictions(S, List, Restrictions, Color)`
   (a) S is the size of the row
   (b) List is the row that is being restricted
   (c) Restrictions is the list of the numbers for the respective row
   (d) Color is the color that is being analized (white if it is a row/ black if it is a column)
   (e) domain is between 0 and 1

   - Restricts a single row. Domain is between black or white, 1 and 0 respectively.
3. `solutions(Rcolumns, Rrows, Rows)`
   (a) Rcolumns is the list of lists of restrictions for each column
   (b) Rrows is the list of lists of restrictions for each row
   (c) Rows is the solved board
   (d) domain is between 0 and 1

   - Finds the solution for the problem by applying the restrictions.

3.2 Constraints

The following constraints are applied:

1. *List of numbers* - A number restriction is when there is a number or more on a column or row. These constraints are applied by `apply_restrictions` which summons `apply_merged` and `apply_single_merged`. `apply_restrictions` calls the other two until there are no more rows or columns to restrict. The two paint the cells in the right indexes (found with`getPossibilities`) with the given color and paint the rest of the row the opposite color afterwards.

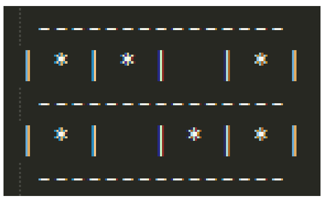
2. *Empty list* - An empty list means there are no cells with that color on the row. If `apply_restrictions` is called with an empty list of restrictions it calls `swap_color` and `color_all` which color the entire row with the opposite color.
3. *No restriction* - If a row has no restrictions any combination is possible. This is represented by [x]. If this happens, `apply_restrictions` does not have any effect.

3.3 Search Strategy

The labeling is called with an empty list of options (same as [leftmost,step,up,all]). This happens because of the problem which does not need any specific method to restrict the board.

3.4 Clpfd built-in predicates

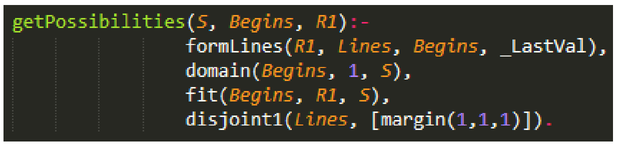
The predicate `getPossibilities` uses the disjoint1 clpfd predicate. After running it, Begins is a list of domain variables which represent the several indexes for positioning the cells according to the given restriction. For example, for a row with size 4 and restrictions [2,1] the predicate will use disjoint1 to realize the valid combination of positions (Figure 3). The predicate `formLines` helps make lines in the format f(Sj,Dj). Sj represents the domain variable and Dj is the length of the line which is found in the restriction list. The option given to `disjoint1` is margin in order to impose a distance of 1 between each of the lines.



```
 ---------------
| * | * |   | * |
 ---------------
| * |   | * | * |
 ---------------
```

Fig. 3. Possible positions for a column with restriction [2,1].

This predicate does not call labeling in order to keep Begins as a domain variable which is useful for the `solutions` predicate. This predicate uses `getPossibilities` to find the combination of labeled variables that suits every single row and column. The Rrows variable will hold the list of lists labeled with the combination that fulfilled all the restrictions.

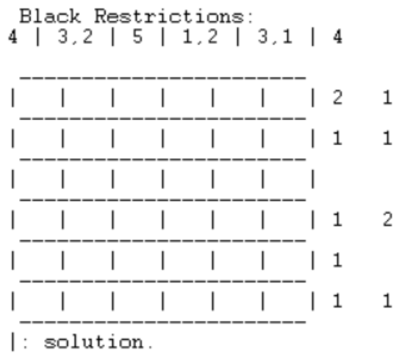


```
getPossibilities(S, Begins, R1):-
                formLines(R1, Lines, Begins, _LastVal),
                domain(Begins, 1, S),
                fit(Begins, R1, S),
                disjoint1(Lines, [margin(1,1,1)]).
```

Fig. 4. The getPossibilities predicate.

4 Solution Presentation

Initially the board is displayed as an empty board (Figure 5) with the restrictions on the top and right.



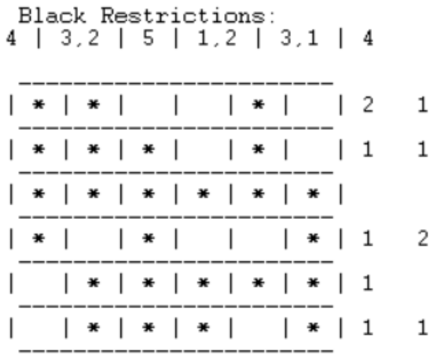
```
 Black Restrictions:
4 | 3,2 | 5 | 1,2 | 3,1 | 4

 -------------------------
|   |   |   |   |   |   | 2   1
 -------------------------
|   |   |   |   |   |   | 1   1
 -------------------------
|   |   |   |   |   |   |
 -------------------------
|   |   |   |   |   |   | 1   2
 -------------------------
|   |   |   |   |   |   | 1
 -------------------------
|   |   |   |   |   |   | 1   1
 -------------------------
|: solution.
```

Fig. 5. Display of an empty board of size 6.

After writing on the console 'solutions.' the solutions predicate is called and the solved board is displayed on the console (Figure 6). The simbol ' * ' stands for a painted cell and the empty spaces stand for white cells.

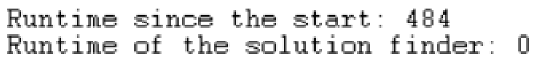
The commands allowed for the user are either 'solutions' or 'end' or 'exit'. The first shows the solution and the two last abort the execution. After the solution is displayed the runtime can be consulted (Figure 7).

The `displayBoard` predicate is responsible for drawing the board and the white restrictions. It calls `displayLine` which calls `countLines` and `writeLine`. These draw the lines that look like this: '——'. Afterwards, the `writeDashes` predicate is called. It resorts to other predicates such as `displayBoardElem` which gets the board element to display and writes it between the '—'. Following this, the `writeWhiteRes` predicate is called. Resorting to `writeElement`, it writes the numbers of the resctrictions for the white squares.



```
 Black Restrictions:
4 | 3,2 | 5 | 1,2 | 3,1 | 4

 -----------------------
| * | * |   |   | * |   | 2   1
 -----------------------
| * | * | * |   | * |   | 1   1
 -----------------------
| * | * | * | * | * | * |
 -----------------------
| * |   | * |   |   | * | 1   2
 -----------------------
|   | * | * | * | * | * | 1
 -----------------------
|   | * | * | * |   | * | 1   1
 -----------------------
```

Fig. 6. Display of the solution.



```
Runtime since the start: 484
Runtime of the solution finder: 0
```

Fig. 7. Display of the runtime.

5 Results

The runtime is only affected when the size of the board is 7 or higher. Any size up to 7 gives a medium runtime of zero. After 7 the runtime rises exponentially. The runing time also depends on the problem. There are problems that prove to be more of a challenge and the restrictions take longer to implement. This leads to different runtimes in boards of the same size. The problems with restriction's numbers as high as the board size are easier to solve. For example for a board of size 10 if a line has a restriction of 10 the runtime is almost zero. This would differ from the time it takes to find the solution for a board of the same size whithout such a restriction. One of the hardest tested problems gave a runtime of 21000. The puzzles found on the website are resolved with a runtime of zero independently of the size. The random puzzles can have higher difficulties which leads to higher runtimes. After the size 10, with random puzzles, the runtime regists values higher than 41000. As seen in Figure 8 the values rise abruptly after size 9. The line on the background represents the trendline for the runtimes.

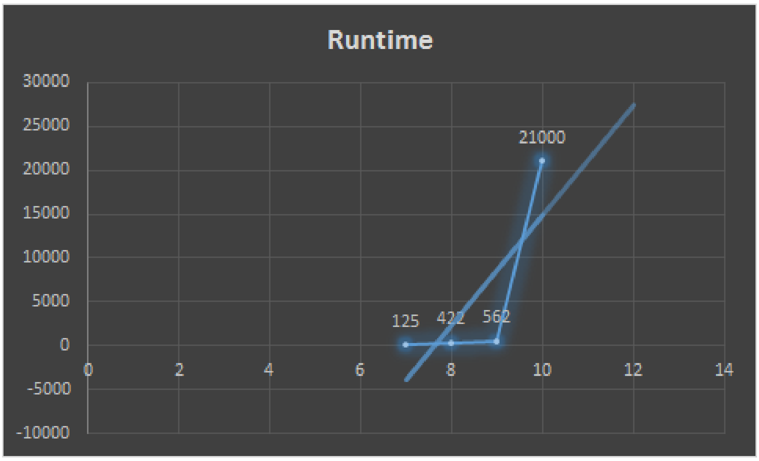


Fig. 8. Graphic featuring the highest runtimes obtained.

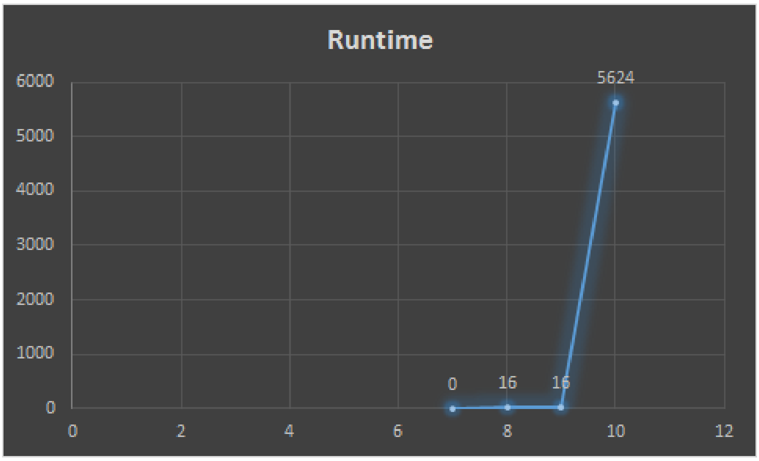


Fig. 9. Graphic featuring the lowest runtimes obtained.

6 Conclusions and Future Work

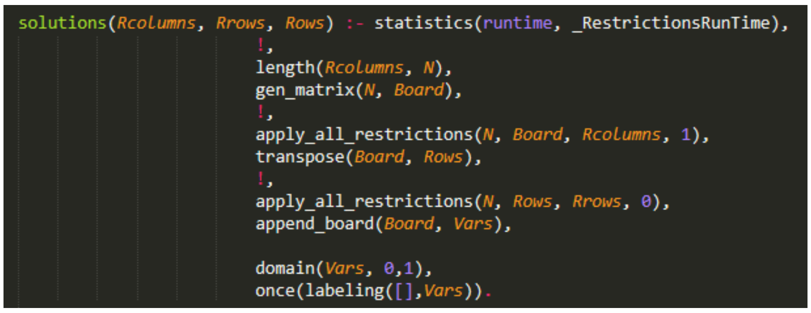
To sum up, the results prove to be in a certain way inconclusive since the runtime changes according to the puzzle. This could be due to many reasons which difficults the task of boosting the efficiency. The dynamic creation of the board could be bettered (instead of being random).

The group feels that this project had a very high level of difficulty. The implementation proved to be a gruelling task and problems arised as the problem was deepened. Despite that, the developed solution fulfilled the objectives although it could be bettered. The efficiency is also an issue to dive into. Considering all paradigms, the group is satisfied with the developed work and looks forward to apply the gained experience and knowledge as a programmer.

References

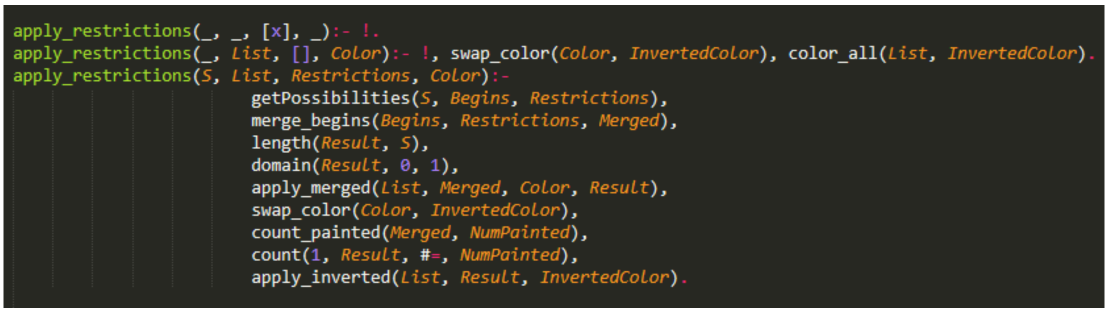
1. Sicstus Prolog Manual, https://www.kth.se/polopoly_fs/1.339598!/sicstus.pdf
2. Clpfd Documentation, https://sicstus.sics.se/sicstus/docs/4.1.0/html/sicstus/lib_002dclpfd.html
3. Clpfd Manual, https://sicstus.sics.se/sicstus/docs/3.7.1/html/sicstus_33.html

7 Annex



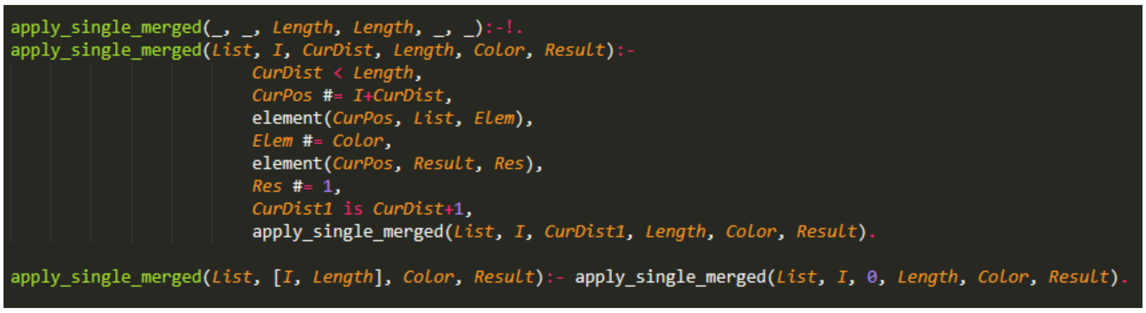
```
solutions(Rcolumns, Rrows, Rows) :- statistics(runtime, _RestrictionsRunTime),
                                    !,
                                    length(Rcolumns, N),
                                    gen_matrix(N, Board),
                                    !,
                                    apply_all_restrictions(N, Board, Rcolumns, 1),
                                    transpose(Board, Rows),
                                    !,
                                    apply_all_restrictions(N, Rows, Rrows, 0),
                                    append_board(Board, Vars),

                                    domain(Vars, 0,1),
                                    once(labeling([],Vars)).
```

Fig. 10. The solution predicate.



```
apply_restrictions(_, _, [x], _):- !.
apply_restrictions(_, List, [], Color):- !, swap_color(Color, InvertedColor), color_all(List, InvertedColor).
apply_restrictions(S, List, Restrictions, Color):-
                        getPossibilities(S, Begins, Restrictions),
                        merge_begins(Begins, Restrictions, Merged),
                        length(Result, S),
                        domain(Result, 0, 1),
                        apply_merged(List, Merged, Color, Result),
                        swap_color(Color, InvertedColor),
                        count_painted(Merged, NumPainted),
                        count(1, Result, #=, NumPainted),
                        apply_inverted(List, Result, InvertedColor).
```

Fig. 11. The predicate that applys restrictions.



```
apply_single_merged(_, _, Length, Length, _, _):-!.
apply_single_merged(List, I, CurDist, Length, Color, Result):-
                        CurDist < Length,
                        CurPos #= I+CurDist,
                        element(CurPos, List, Elem),
                        Elem #= Color,
                        element(CurPos, Result, Res),
                        Res #= 1,
                        CurDist1 is CurDist+1,
                        apply_single_merged(List, I, CurDist1, Length, Color, Result).

apply_single_merged(List, [I, Length], Color, Result):- apply_single_merged(List, I, 0, Length, Color, Result).
```

Fig. 12. The predicate that restricts each line.