Solving the Puzzle Magnets with Constraint Logic Programming

Ângela Cardoso and Nuno Valente

FEUP-PLOG, Turma 3MIEIC02, Grupo Magnets_2.

Abstract. This project was developed as part of the coursework for the subject of Logic Programming at the Faculty of Engineering of the University of Porto. In order to learn logic programming with constraints, we chose the puzzle Magnets to implement in SICStus Prolog. The application obtained is capable of solving a given Magnets puzzle, as long as it is supplied in a specific format, generating random puzzles of the required size and displaying both the puzzle and the solution in a user friendly text format. The efficiency of the solution we obtained depends, of course, on the size of the puzzle, but more so on its degree of difficulty. One can easily obtain puzzles where the constraints do not allow for immediate variable reduction, which leads to frequent backtracking and thus long execution times. There is also a lot of symmetry in theses puzzles, which in a straightforward implementation often increments the difficulty of reaching a solution.

1 Introduction

Constraint logic programming is very useful to solve problems in which one wants to determine a set of variables that must satisfy certain rules. In addition to the the logic programming literals, these programs contain constraints such as $X \neq Y$. The goal of the program is to determine values for all the variables such that all the clauses and constraints are satisfied. Typically, this kind of programming is used to solve search problems, such as paper-and-pencil puzzles or chess puzzles, and optimization problems, such as scheduling, timetabling, resource allocation, planning, production management or circuit check. By using constraint logic programming one can reduce the development time, as well as obtain efficient, simple and easy to maintain solutions.

To solidify our knowledge of constraint logic programming, we chose to implement the paper-and-pencil puzzle Magnets. In this puzzle, one is given a square grid of dominoes, as well as the number of positive and negative poles for each row and column. Some dominoes are magnetized, having a positive pole and a negative pole, while others are neutral. Also, poles with the same charge repel and are not allowed to touch vertically or horizontally. The objective is identify the positive poles with a $+$ sign, the negative poles with a $-$ sign and the neutral ones with an x.

The following sections describe the progress of our work on this subject, with details of our implementation and of the results we obtained, including statistics that give some insight to the quality of our solution.

2 Problem Description

Although we managed to find several online implementations or descriptions of the Magnets puzzle, we could not determine its origins. In any case, the rules are quite clear:

– for each row and each column the number of positive and negative poles must be as specified;
– each charged domino must have a positive and a negative pole;
– two positive, or two negative, charged poles must not be vertically or horizontally adjacent.

Figure 1 illustrates these rules, showing a solved Magnets puzzle.

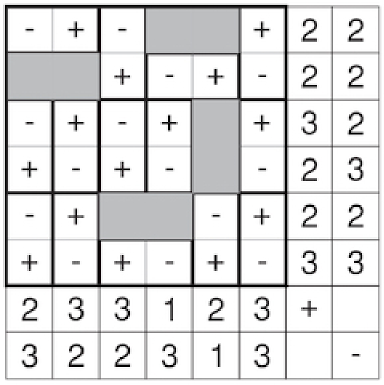


Fig. 1: Example of a Magnets puzzle

Our objective is to use constraint logic programming to implement a Magnets puzzle solver. Theoretically, our implementation should be able to handle any such puzzle. However, the exponential nature of such puzzles will certainly impose limitations on any devised implementation. Hence, it is also our intent to obtain an efficient method to solve these puzzles.

3 Approach

Even a casual observation of the rules of Magnets gives some clue that this puzzle is suited for constraint logic programming. We shall make this clear in the following sections, where we present a formal description of Magnets as a constraint satisfaction problem (CSP).

3.1 Decision Variables

In any CSP there are variables whose value we aim to determine. For Magnets, the decision variables are the half dominoes or poles, that is, each small square in the grid is a decision variable. In order to solve the puzzle, we must obtain the value (positive, negative or neutral) of each of these poles. To better represent these values, and in order to allow for constraint resolution in SICStus Prolog, we used the following conversions:

- 1 represents positive poles;
- -1 represents negative poles;
- 0 represents neutral poles.

Since Magnets puzzles are square grids of a given size N, the number of decision variables is N^2. Each of these variables has 3 possible values, therefore there is a total of 3^{N^2} possible solutions to be tested. These variables are organized sequentially in a list from top of the grid to bottom and from left to right.

3.2 Constraints

For each Magnets rule, there is a rigid constraint in our Magnets CSP. The first restriction is on the number of positive and negative poles for each row and column. In order to implement this in SICStus Prolog, we used the combinatorial constraint `global_cardinality`. First we organize our puzzle as a list of rows, then we use `global_cardinality` to force each row to have a specific number of $1s$, $0s$ and $-1s$. Afterwards, the same thing is done for the columns.

The second rule forces each charged domino or magnet to have a positive and a negative pole. This is done with a simple arithmetic constraint. Given our codification of positive, negative and neutral, the sum of both poles of a magnet must be zero. Hence, we use the constraint `#=` to enforce this equality.

Finally, there is a rule on the proximity of same charge poles. Two positive (respectively, negative) poles repel each other, therefore, they cannot be vertically or horizontally adjacent. Again, we check this with an arithmetic constraint. Given two neighbor (vertically or horizontally adjacent) poles, the sum of their cannot be bellow -1 or above 1. Indeed, we can only obtain 2 if there are two positive adjacent poles, and the same thing for -2 and negative poles. Hence, we use the constraints `#>` and `#<` to verify this last rule.

3.3 Search Strategy

In order to attribute values to the variables, SICStus uses the predicate labeling, for which one may provide some options that, depending on the problem at hand, may increase the efficiency of the search for a solution. In our implementation of Magnets, these options where mainly chosen through trial and error. In fact, we started by using no options at all (which is the same as using the default options) and then, with a collection of somewhat difficult puzzles, we determined the options with better time performance.

The first option to consider is variable ordering. That is, which variable should be first consider when Prolog tries to find a possible solution. The default option starts with the leftmost variable, in our case, the first square of the grid. There are several other options that we will not exhaustively mention, because we quickly concluded that they are not suitable for our problem. Besides `leftmost` the ones we tested where:

– `first_fail`, which chooses the leftmost variable with the smallest domain;
– `occurrence`, which chooses the leftmost variable with the largest number of constraints;
– `ffc`, which combines first `first_fail` and `occurrence`.

The performance of these options is not always consistent, that is, the better option depends on the specific example. However, in our limited experiments, there seems to be an advantage for `leftmost`, even though sometimes `first_fail` is better. Ordering options including `occurrence` did not perform well in our examples. This is probably due to the fact that the number of restrictions on each variable is more or less the same, with the exception being the extremes of the puzzle, since they have less neighbors. In the end, we opted to use the default `leftmost`, because there is some limited evidence of its superiority.

The next option is about value selection. Given a variable and a set of possible values for that variable, how should Prolog decide which one to try first. We also made some experiments with these options, but we quickly decided on the best one. Indeed, there are at most three possible values for a Magnets variable: -1, 0 and 1. For every -1 there is a 1, because of the second rule, that is, the total number of $-1s$ is the same as the total number of $1s$. On the other hand, $0s$ come in pairs, since every neutral domino must have two of them. This means that in a puzzle where there is some equilibrium between the number of neutral and charged dominoes, there will be approximately twice as much $0s$ as $-1s$ or $1s$. This means that the average randomly generated puzzle has more $0s$ and thus, we are more likely to succeed if we try 0 first. With this reasoning in mind we chose the option `median` for value selection, since for every variable with domain $\{-1, 0, 1\}$ this will lead to 0 being chosen first.

Once 0 has been tested, given the puzzle symmetry, there is no advantage in trying -1 or 1 first, therefor we chose no option for value ordering, which means that the default `up` is used.

To test these options we ran the same 1000 random puzzles of size 10, with different sets of options. The average time using the default options was 25 seconds, with `first_fail` the time was 28 seconds, with `median` 23 seconds, and with both `first_fail` and `median` 36 seconds. Although these results have very slight variations, the maximum time is quite more telling, with 1410 seconds for the defaults, 3360 seconds for `first_fail`, 870 seconds for `median` and 5360 seconds for both `first_fail` and `median`. These were the results that influenced our decision in the final application.

4 Puzzle Generation

In order to test our Magnets solver, we decided to implement a random Magnets puzzle generator. Given a positive even integer, this generator creates a random Magnets puzzle of that size, for which there is at least one solution. As it turns out, this is also an adequate problem to solve with constraint logic programming. Indeed, to have a valid Magnets puzzle, we need a square grid of dominoes, as well as the number of negative and positive poles in each row and each column. As in the previous section we will now formalize the description of Magnets puzzle generation as a CSP, but we will do so in a more succinct manner. In particular, we used no labeling options other than the default, so there is no subsection about that.

4.1 Decision Variables

In order to obtain a Magnets puzzle, first we need the square grid of dominoes. In our implementation we represent each cell of the grid with a cardinal point (`n`, `s`, `e` or `w`), that points to the other half of its domino. Thus, the list of half dominoes forms our first set of variables. Since SICStus Prolog demands small integer domains constraint resolution, our program actually uses integers from 1 to 4 to represent the cardinal points, but after these variables are assigned they are translated to the letters above.

Besides the grid of dominoes, Magnets also has lists with the numbers of positive and negative poles for each row and column. These lists form the rest of our decision variables. If the puzzle has size N, then we have four lists, each with N integers whose domain is $[0, N]$. Thus, in total we have N^2+4N variables to generate a Magnets puzzle of size N.

4.2 Constraints

The first constraints we need to consider are those that allow us to obtain a suitable set of dominoes. These dominoes cannot overlap and they must fill the entire square. If we starting from the top left corner, going to the right and then to the bottom, it is not hard to guarantee the no overlap rule. All we have to do is, for each new domino, chose either the east or the south direction. Unless we are at a border, then only one of these is available.

However, as we get closer to the bottom of the square, we may find ourselves trapping a single cell, which contradicts the rule of filling the whole grid with dominoes. To overcome this, we still let Prolog chose randomly between east and south for new dominoes, but we add a constraint that states that for each half domino has a corresponding half with opposite direction. Since all cells of the grid must have a direction, Prolog then uses backtracking to verify this constraint and obtain a suitable grid.

The real trick here is to chose the next value randomly, instead of with some prefixed order has is with all labeling options. To obtain this random effect, we added redundant constraints and used random asserts and retracts to establish the order in which the satisfaction of these constraints is attempted.

After obtaining the grid of dominoes, we fill that grid with the values -1, 0 and 1, applying the same constraints from the puzzle solution, that is, the sum of both poles of a magnet must be zero and the sum of two vertically or horizontally adjacent poles is at least -1 and at most 1. Once this is done, we compute the number of negative and positive poles in each row and column by simply counting them.

5 Solution Presentation

In addition to random puzzle generation and puzzle resolution, we also implemented some Prolog predicates that allow us tu visualize the puzzles and the solutions in text mode. In fact, we have a very small application that asks the user for the size of the puzzle, shows a random puzzle of that size and then, when the user choses to, presents the solution.

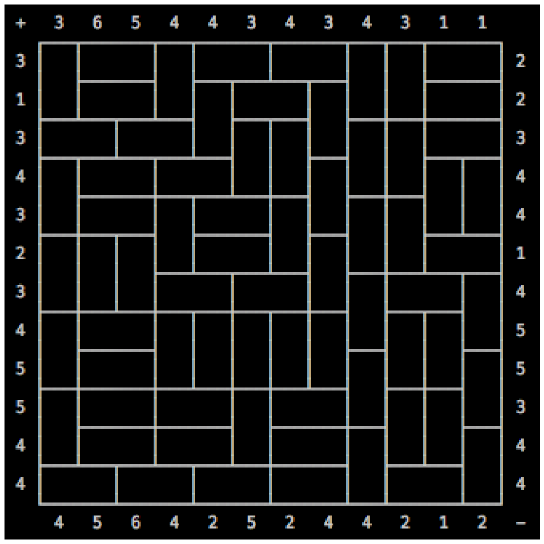
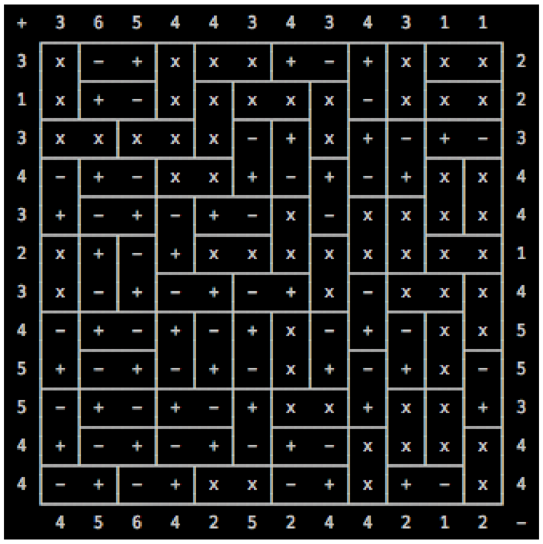


Fig. 2: The display of a Magnets puzzle

Figure 2 shows the output of the puzzle display predicate for a size 12 puzzle, while Figure 3 shows the solution of the same puzzle.



```
+  3  6  5  4  4  3  4  3  4  3  1  1
3  x  -  +  x  x  x  +  -  +  x  x  x  2
1  x  +  -  x  x  x  x  x  -  x  x  x  2
3  x  x  x  x  x  -  +  x  +  -  +  -  3
4  -  +  -  x  x  +  -  +  -  +  x  x  4
3  +  -  +  -  +  -  x  -  x  x  x  x  4
2  x  +  -  +  x  x  x  x  x  x  x  x  1
3  x  -  +  -  +  -  +  x  -  x  x  x  4
4  -  +  -  +  -  +  x  -  +  -  x  x  5
5  +  -  +  -  +  -  x  +  -  +  x  -  5
5  -  +  -  +  -  +  x  x  +  x  x  +  3
4  +  -  +  -  +  -  +  -  x  x  x  x  4
4  -  +  -  +  x  x  -  +  x  +  -  x  4
   4  5  6  4  2  5  2  4  4  2  1  2  -
```

Fig. 3: The display of a solved Magnets puzzle

The predicates used to obtain these outputs just go through the puzzle and the result, both represented as lists of rows, and draw the grid and its contents. The hardest part is to use the correct box drawing characters depending on the position of each domino.

6 Results

We used our random puzzle generator and some of SICStus statistics predicates to test the efficiency of the solver. Since the time varies a lot between different puzzles of the same size, we computed, in addition to the average run time, the minimum and the maximum. We also tracked the average number of backtracks for puzzle solving. The results are presented in Table 1, with the time measured in seconds.

Puzzle Size	2.00	4.00	6.00	8.00	10.00	12.00
Minimum Time	0.00	0.00	1.00	2.00	3.00	6.00
Average Time	0.34	1.08	2.26	5.27	19.90	865.47
Maximum Time	6.00	13.00	28.00	81.00	353.00	20609.00
Average Backtracks	0.00	0.16	0.94	7.54	207.64	13998.90

Table 1: Magnets solution efficiency

Using a logarithmic scale of base 3, which is appropriate since everything points to an exponencial time complexity, we obtained from Table 1 the graph in Figure 4, where the puzzle sizes range from 2 to 12.

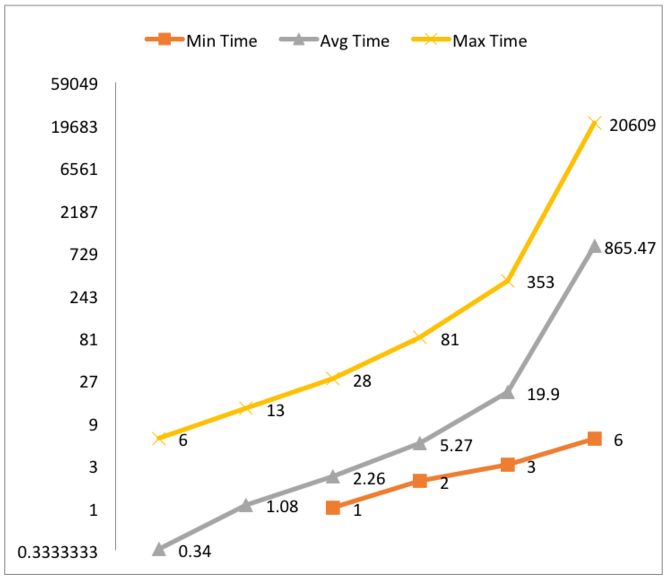


Fig. 4: Time to solve a Magnets puzzle

Due to time constraints, these tests where performed in samples of 100 puzzles and we could not finish the test for 100 puzzles of size 14. But with the time differences observed in same size puzzles, we do not put much trust in these results. More robust tests should be performed in a powerful machine that could be left running the puzzles for a long time. That way, it would be possible to obtain bigger samples and with larger sizes, leading to an experimental conjecture for time complexity of our solver.

7 Conclusions and Future Work

This project allowed us to better understand constraint logic programming and how well suited it is for puzzle solving. In fact, the use of Prolog and in particular of constraints made this task much simpler than it would have been with other languages or tools.

In the end our implementation is very close to the rules of the puzzle Magnets, with only some translations and improvements that allow us to avoid symmetry. We did try to think of other redundant constraints that could improve the results. However none of our ideas has materialized. This would be one of the lines for continuing this work, to use a more sophisticated constraint program that can reduce the time to compute a solution.

Even tough our tests where limited, the results seem to point to exponential complexity, in fact the theoretical number 3^{N^2} is in agreement with the run times we obtained experimentally. Of course, further tests should be conducted, with bigger samples and larger puzzles, in order to comprove these initial results.

Depending on the puzzle, our solver can reach a solution quite fast, even for larger puzzles, but the fact that the time varies so much makes it somewhat unreliable. However, for the casual user, with puzzles of size up to 10, the application should have no problem in computing the solution in a reasonable amount of time. At sizes 14 and above its a trial and error procedure, some times quite fast and other times excruciatingly slow.

A Source Code

A.1 File display.pl

```prolog
/* -*- Mode:Prolog; coding:utf-8; -*- */

/***************/
/* print board */
/***************/

/* predicate used to print the game board */
printBoard(I) :-
        nl , printNumbers ,                                /* print the numbers on top of the board */
        write('          ') , printPlayerName(I) , nl ,     /* print last player name */
        printGridTop , nl ,                                /* print the grid top */
        printRows(I, 1) ,                                  /* print the board rows */
        printNumbers , nl , nl , !.                        /* print the bottom numbers */

/* predicate used to print the game board */
printBoard :-
        nl , printNumbers , nl ,                           /* print the numbers on top of the board */
        printGridTop , nl ,                                /* print the grid top */
        printRows(1) ,                                     /* print the board rows */
        printNumbers , nl , nl , !.                        /* print the bottom numbers */
```

```
/* predicate used to print the rows of the board */
printRows(_, 19) :- !.
                                    /* stop after row
   18 */
printRows(I, X) :-
        printLeftNumbers(X) ,
                                     /* print number
           on the left of the row */
        printCells(X, 1) ,
                                        /* print row
           cells */
        printRightNumbers(X) ,
                                    /* print numbers
           on the right of the row */
        (even(X) ->
         O is div(X, 2) ;
         (Y is X + 1 ,
          O is div(Y, 2))) ,
        (even(X) ->
                                                /* if
           the row number is even */
          printLastPlayNumbers(I, O) ; !) ,
                         /* print lastPlay piece
             numbers */
        nl , printGrid(X) ,
                                       /* print grid
           line */
        (even(X) ->
                                                /* if
           the row number is even */
         printLastPlayBottom(I, O) ;
                              /* print lastPlay piece
             bottom */
         printLastPlayTop(I, O)) ,
                                /* else print
            lastPlay piece top */
        X1 is X + 1 , nl ,
                                        /* move on to
           next row */
        printRows(I, X1).
                                         /* calling
           the function recursively */

/* predicate used to print the rows of the board */
printRows(19) :- !.
                                       /* stop after
   row 18 */
printRows(X) :-
```

```
        printLeftNumbers(X) ,
                                        /* print number
           on the left of the row */
        printCells(X, 1) ,
                                           /* print row
           cells */
        printRightNumbers(X) , nl ,
                                     /* print numbers on the
            right of the row */
        printGrid(X) ,
                                                 /* print
           grid line */
        X1 is X + 1 , nl ,
                                           /* move on to
           next row */
        printRows(X1).
                                                 /* calling
            the function recursively */

/* predicate used to print the cells of a row of the
   board */
printCells(_,19) :- !.
                                        /* stop after cell
    18 */
printCells(X, Y) :-
        getTopLevel(X, Y, L) ,
                                        /* determine the
           current level of the cell */
        (L > 0 ->
                                                      /* if
            the current level is above 0 */
         (halfPiece(X, Y, L, N, C) ,
                                     /* determine what half
            piece is at that top level */
          print(N) ,
                                                     /*
             print the number of the half piece */
          printCardinal(C) ,
                                           /* print the
             cardinal of the half piece */
          print(L)) ;
                                                    /*
             print the level of the half piece */
         write('   ')) ,
                                                /*
            otherwise, if the level is 0, print 3
            spaces */
        write('|') ,
                                                   /* in
           any case, print the cell grid divider */
        Y1 is Y + 1 ,
                                                    /*
           advance to the next cell on the same row */
        printCells(X, Y1).
                                               /* calling the
            function recursively */

/* predicate used to obtain the top level of a given
   position */
getTopLevel(X, Y, 0) :- not(halfPiece(X, Y, _, _, _)).
    /* if there is no half piece on that position the
   level is 0 */
getTopLevel(X, Y, L) :-
                                          /* otherwise, the
   level is L if */
        halfPiece(X, Y, L, _, _) ,
                                     /* there is an half
           piece on that position on level L */
        L1 is L + 1 ,
        not(halfPiece(X, Y, L1, _, _)).
                                /* and there is no half
           piece on that position on level L + 1 */

/* predicate used to print the cardinal of an half
   piece in an appealing way */
printCardinal(n) :- write('↑').
printCardinal(e) :- write('→').
printCardinal(s) :- write('↓').
printCardinal(w) :- write('→').

/* predicate used to print the numbers on top of the
   board */
printNumbers :-
        write('            1   2   3   4   5   6   7
           8   9  10  11  12  13  14  15  16  17  18')
           .

/* predicate used to print the numbers on the left of
   the board */
printLeftNumbers(X) :-
        ((X < 10,
                                                        /*
           for numbers before 10, there is one extra
           space */
          write('         ') ,
          print(X) ,
          write('|')) ;
                                                    /* also
             print the board grid divider */
         (write('        ') ,
          print(X) ,
          write('|'))).

/* predicate used to print the numbers on the right of
   the board */
printRightNumbers(X) :-
        ((X < 10 ,
          write(' '),
          print(X)) ;
         (print(X))).

/* predicate used to print the top of the board grid */
printGridTop :- write('              ┌───┬───┬───┬───┬───┬───┬─
   ──┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┐
   ').

/* predicate used to print the board grid */
printGrid(X) :-((X < 18 ,
                              /* the last row is
   different */
                 write('          ├───┼───┼───┼───┼───┼───┼─
                    ──┼───┼───┼───┼───┼───┼───┼───┼───┼
                    ───┼───┼───┼───┤')) ;
                 write('          └───┴───┴───┴───┴───┴───┴─
                    ──┴───┴───┴───┴───┴───┴───┴───┴───┴
                    ───┴───┴───┴───┘')).


/******************/
/* print last play */
/******************/

printLastPlayTop(I, O) :-
        lastPlay(_, _, O, I) -> write('       ┌───┬───┐
           ') ; !.


printLastPlayBottom(I, O) :-
        lastPlay(_, _, O, I) -> write('       └───┴───┘
           ') ; !.

printLastPlayNumbers(I, O) :-
        lastPlay(N1, N2, O, I) ->
        (write('     | ') ,
                                    /* print left
           border */
         print(N1),
                                             /*
            print left number */
         write(' | ') ,
                                        /* print
            center divider */
         print(N2) ,                                          /*
            print right number */
         write(' |')) ; !.                              /* print
            right border */


/***************/
/* print player */
/***************/

/* predicate used to print N spaces */
printSpaces(N) :-
        N == 0 -> ! ;                                       /* when N
            = 0, stop */
        (write(' ') ,                                       /*
           otherwise write a space */
         N1 is N - 1 ,                                      /*
            decrease N */
         printSpaces(N1)).                              /* and repeat
             */

/* predicate used to print a player's name  taking
   exactly 9 characters */
printPlayerName(I) :-
        player(I, S, _) ,                               /* find the
           player name S, S has at most 8 characters
           */
        print(S) ,                                             /*
           print the player name */
        atom_length(S, N) ,                           /* determine
           the name's length */
        M is 9 - N ,                                         /* M is
           how many characters are missing to 9 */
        printSpaces(M).                                   /* print M
           spaces */

/* predicate used to print a player's name and pieces,
   with starting spaces */
printPlayer(I) :- write('         ') , printPieces(I,
   0, 0, 0, 0, 0, 0, 0, 0).
```

```
/* predicate used to print a player's name and pieces
   */
/* this is a somewhat complex function and we do not
   present step by step comments.
   alternatively, here is the idea:
   - the pieces are displayed in two rows;
   - each piece has top, numbers and bottom;
   - so each piece row is in fact 3 rows: top, numbers
     and bottom;
   - to know what to print in each moment, we keep
     track of the current row R;
   - if R is 0 or 3, we are printing tops, for 1 and 4
     we print numbers and the others are bottoms;
   - row 1 also has the player name;
   - we also need to keep track of the current piece
     numbers N1 and N2;
   - and we need to do it for all three aspects, top,
     number and bottom;
   - so we have N1T, N2T, N1N, N2N, N1B, N2B;
   - if a piece has already been played, we print
     spaces in its place;
   - this way, the pieces stay in the same place from
     beggining to end;
   - to determine if we should change row, we check if
     the counter C has reached 9;
   - because each row has 9 pieces;
   - we use specific predicates to print tops, numbers
     and bottoms */
printPieces(I, C, R, N1T, N2T, N1N, N2N, N1B, N2B) :-
        (C == 9 -> (nl , (R == 1 -> printPlayerName(I)
           ; write('          ')) ,
                  C1 is 0 , R1 is R + 1 , printPieces
                     (I, C1, R1, N1T, N2T, N1N, N2N,
                      N1B, N2B)) ;
         ((R == 0 -> (N1T > 7 -> ! ;
                      (N2T > 7 -> (N2T1 is N1T + 1 ,
                         N1T1 is N1T + 1 , printPieces
                         (I, C, R, N1T1, N2T1, N1N,
                         N2N, N1B, N2B)) ;
                       ((piece(N1T, N2T, I, P) -> C1 is
                           C + 1 , printPieceTop(P) ;
                          C1 is C),
                        N2T1 is N2T + 1 , printPieces(I
                           , C1, R, N1T, N2T1, N1N,
                           N2N, N1B, N2B)))) ;
           ((R == 1 -> (N1N > 7 -> ! ;
                        (N2N > 7 -> (N2N1 is N1N + 1 ,
                           N1N1 is N1N + 1 ,
                           printPieces(I, C, R, N1T,
                           N2T, N1N1, N2N1, N1B, N2B))
                            ;
                         ((piece(N1N, N2N, I, P) -> C1
                             is C + 1 ,
               printPieceNumber(N1N, N2N,
                P) ; C1 is C),
             N2N1 is N2N + 1 , printPieces
                (I, C1, R, N1T, N2T, N1N,
                 N2N1, N1B, N2B)))) ;
((R == 2 -> (N1B > 7 -> ! ;
             (N2B > 7 -> (N2B1 is N1B + 1
                , N1B1 is N1B + 1 ,
                printPieces(I, C, R, N1T,
                 N2T, N1N, N2N, N1B1,
                N2B1)) ;
              ((piece(N1B, N2B, I, P) ->
                 C1 is C + 1 ,
                 printPieceBottom(P) ; C1
                  is C),
              N2B1 is N2B + 1 ,
                  printPieces(I, C1, R,
                  N1T, N2T, N1N, N2N, N1B
                  , N2B1)))) ;
  ((R == 3 -> (N1T > 7 -> ! ;
               (N2T > 7 -> (N2T1 is N1T +
                  1 , N1T1 is N1T + 1 ,
                  printPieces(I, C, R,
                  N1T1, N2T1, N1N, N2N,
                  N1B, N2B)) ;
                ((piece(N1T, N2T, I, P) ->
                    C1 is C + 1 ,
                   printPieceTop(P) ; C1
                   is C),
                 N2T1 is N2T + 1 ,
                    printPieces(I, C1, R,
                     N1T, N2T1, N1N, N2N,
                     N1B, N2B)))) ;
    ((R == 4 -> (N1N > 7 -> ! ;
                 (N2N > 7 -> (N2N1 is N1N
                    + 1 , N1N1 is N1N + 1
                     , printPieces(I, C,
                    R, N1T, N2T, N1N1,
                    N2N1, N1B, N2B)) ;
                  ((piece(N1N, N2N, I, P)
                     -> C1 is C + 1 ,
                     printPieceNumber(N1N
                     , N2N, P) ; C1 is C)
                     ,
                   N2N1 is N2N + 1 ,
                      printPieces(I, C1,
                      R, N1T, N2T, N1N,
                      N2N1, N1B, N2B))))
                      ;
      ((R == 5 -> (N1B > 7 -> ! ;
                   (N2B > 7 -> (N2B1 is
                      N1B + 1 , N1B1 is
                      N1B + 1 ,
                                                         printPieces(I, C, R
                                                         , N1T, N2T, N1N,
                                                         N2N, N1B1, N2B1)) ;
                                                       ((piece(N1B, N2B, I, P
                                                          ) -> C1 is C + 1 ,
                                                           printPieceBottom(
                                                          P) ; C1 is C),
                                                        N2B1 is N2B + 1 ,
                                                           printPieces(I, C1
                                                           , R, N1T, N2T,
                                                           N1N, N2N, N1B,
                                                           N2B1)))) ;
                                          !))))))))))))).

/* predicate used to print a piece's top */
printPieceTop(P) :-
        P == 0 ->
                                                                      /* if
            the piece has not been played */
        write(' ┌───┬───┐') ;
                                                          /* print top */
        write('          ').
                                                           /* otherwise
           print spaces */

/* predicate used to print a piece's numbers */
printPieceNumber(N1, N2, P) :-
        P == 0 ->
                                                                      /* if
            the piece has not been played */
        (write(' | ') ,
                                                                /* print
           left border */
         print(N1),
                                                                     /*
            print left number */
         write(' | ') ,
                                                                 /* print
            center divider */
         print(N2) ,
                                                                    /*
            print right number */
         write(' |')) ;
                                                                 /* print
            right border */
        write('          ').
                                                           /* otherwise
           print spaces */

/* predicate used to print a piece's bottom */
```

```
printPieceBottom(P) :-
                                    /* very similar to
    the printPieceTop */
        P == 0 ->
        write(' └───┴───┘') ;
        write('          ').


/*************/
/* print game */
/*************/

/* predicate used to print the current board and player
    pieces */
printGame(I1, I2) :-
        cls , printBoard(I2) , printPlayer(I1).

/* predicate used to print the board and the result at
   the end of the game */
printGameOver(I) :-
        cls , printBoard(I) ,
        write('Game Over: ') , player(I, S, _) , print(
           S) , write(' wins!') ,
        nl , nl , sleep(1).
```

References

1. Niederlinski, Antoni: A gentle guide to constraint logic programming via ECL^{i}PS Gliwice 2014
2. Notes from theoretical classes
3. https://sicstus.sics.se/sicstus/docs/3.7.1/html/sicstus_33.html
4. http://www.janko.at/Raetsel/Magnete/index.htm
5. http://www.constraintsolving.com/tutorials/cptutorial
6. http://www.swiprolog.org/man/clpfd.html
7. https://en.wikipedia.org/wiki/Boxdrawing_character