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.

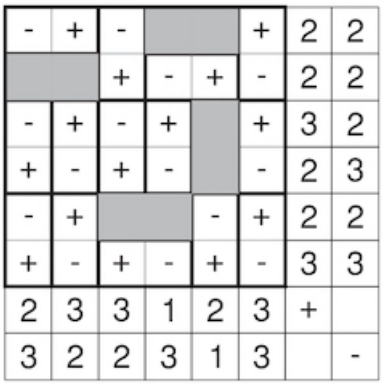

Fig. 1. Example of a Magnets puzzle

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 lines, then we use `global_cardinality` to force each line 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 `first_fail`, even though sometimes `leftmost` 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 `first_fail`, both because it makes theoretical sense and 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.

4 Solution Presentation

Explicar os predicados que permitem visualizar a solucao em modo de texto.

5 Results

Demonstrar exemplos de aplicacao em instancias do problema com diferentes complexidades e analisar os resultados obtidos. Devem ser utilizadas formas convenientes para apresentacao dos resultados (tabelas e/ou graficos).

6 Conclusions and Future Work

Que conclusoes retira deste projeto? O que mostram os resultados obtidos? Quais as vantagens e limitacoes da solucao proposta? Como poderia melhorar o trabalho desenvolvido?

A Source Code

A.1 File display.pl

```
/* -*- 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 */
```

```
20          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) :-
25          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) ;
30           (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 */
35          (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 */
40
    /* 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) ,
```

```
90            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 */
100 printGridTop :- write('          ┌---┬---┬---┬---┬---┬-
   --┬---┬---┬---┬---┬---┬---┬---┬---┬---┬---┬---┐
   ').

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


/*******************/
/* print last play */
110 /*******************/

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

115

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

printLastPlayNumbers(I, O) :-
120     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)))) ;
                          !)))))))))))))).
195
    /* predicate used to print a piece's top */
    printPieceTop(P) :-
            P == 0 ->
                                                    /* if
                the piece has not been played */
            write(' ┌───┬───┐') ;
                                                /* print top */
200         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 */
205         (write(' | ') ,
                                                /* print
               left border */
             print(N1),
                                                     /*
                print left number */
             write(' | ') ,
                                                 /* print
                center divider */
             print(N2) ,
                                                    /*
                print right number */
             write(' |')) ;
                                                 /* print
                right border */
210         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

[1982] Clarke, F., Ekeland, I.: Nonlinear oscillations and boundary-value problems for Hamiltonian systems. Arch. Rat. Mech. Anal. 78, 315–333 (1982)

References

Clarke, F., Ekeland, I.: Nonlinear oscillations and boundary-value problems for Hamiltonian systems. Arch. Rat. Mech. Anal. 78, 315–333 (1982)