Constraint Logic Programming in Prolog: *Hanjie* Puzzle Solver

Luís Cleto and João Marinheiro

FEUP-PLOG, Turma 3MIEIC05, Group 23
`{ei11077,ei11129}@fe.up.pt`
`http://www.fe.up.pt`

Abstract. The purpose of this project was to use constraint logic programming in *Prolog* to implement a solver for the 2D puzzle, *Hanjie*. For this purpose we used the clp(FD) library provided by *SICStus Prolog* 4.2.3, specifically the *sum/3* and *automaton/3* combinatorial constraints. The program we developed is able to solve puzzles with dimensions up to 88x88, with only one possible solution, in less than one second. When there are multiple solutions, the execution time for the obtaining the first solution varies with the number of possible solutions.
These results show that the execution time of the program is primarily affected by the amount of possible results. While larger grid dimensions do increase the execution time, the increase is linear if the number of possible solutions is maintained. On the other hand, increasing the number of possible solutions will lead to an exponential growth in execution time.

1 Introduction

The goal of this project is to use constraint logic programming in *Prolog* to develop a logic program capable of solving a decision problem in the form of the 2D puzzle, *Hanjie*. This puzzle consists of a rectangular grid with 'clues' on top of every column and to the left of every row that indicate the number and length of gray blocks in that column/row. To achieve this goal, first had to be chosen the data structures that would be used to represent the problem. It was decided to use lists of lists, containing the clues for each column/row in every sublist, and a similar structure for the puzzle grid where every sublist is a row of the grid. Every square of the grid can be either a 0 or a 1 where a 0 represents a blank square and 1 represents a gray square. Our implementation also allows the lists of clues to be partially or completely uninstantiated, providing possible values for the uninstantiated elements. Additionally, it allows the puzzle to be written to a file instead of to the terminal as some solutions may be too large to be displayed correctly on the terminal.
The most complex aspect of creating constraints for this puzzle is creating a set of rules that will ensure that every gray block occurs in sequence on the proper row, with the proper dimensions, without bordering any other blocks in that row. To solve this particular problem of sequences, it was opted to use *automaton/3*

constraints to establish the main rules of the puzzle and ensure the integrity of the gray blocks in every row and column.
In addition, a subgoal of the project was to allow for random generation of valid *hanjie* puzzles, which was achieved with an algorithm similar to how a human being would create the puzzle. The grid is created first by randomly filling it with painted squares and the clues for each row/column are obtained afterwards, this way the puzzle is guaranteed to have at least one possible solution.
The remainder of this article will contain several sections dedicated to describing the problem, the data files containing examples of the problem to solve, the decision variables used, the constraints implemented, the adopted search strategy, the solution presentation, the results obtained and the conclusion we achieved. Additionally it will contain in the annex the source code developed for the project.

2 Problem Description

Hanjie is a two dimensional puzzle that starts with a grid of blank squares with clues on top of every column and to the left of every row. Every list of clues contains numbers that indicate the length of every painted block in that row. The length of the list will be the number of blocks that row must contain and every block must be separated from the others by at least one blank square. Additionally, the order of the clues is the order of the blocks in that row and each block must be separated from the next by at least one blank block. For example, a set of clues {3, 2} would indicate that there would have to be exactly two gray blocks, the first composed of three gray squares and the second composed of two gray squares, in that row. They would also have to be separated by at least one blank square. See Fig. 1 for an example of a solved *Hanjie* puzzle.

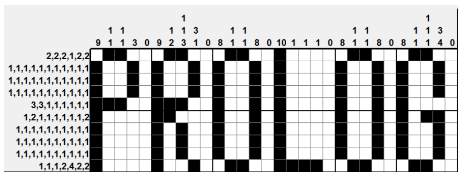


Fig. 1. An example of a *Hanjie* puzzle. This particular example has more than one possible solution.

3 Data Files

Along with the *Prolog* source code developed to solve the described problem, an additional file called 'hanjie_examples.pl'was created containing several examples of *Hanjie* puzzles in the form of *example_name(-ColumnHints, -RowHints)*

where the arguments of the predicate will be initialized with the clues for the columns and the rows in the form of a list of lists for both of them.
There are several examples of puzzles and a few with partially or completely uninstantiated hints for the columns/rows.
Aditionally, there are predicates to randomly generate *Hanjie* puzzles of any dimension in the file 'hanjie_generation.pl'. The predicate *generate_hanjie(+Width, +Height, -ColumnHints, -RowHints)*, to randomly generate a puzzle that may have more than one solution by randomly painting some cells of the grid and generating the clues, *generate_hanjie_full(+Width, +Height, -ColumnHints, -RowHints)* that generates a puzzle with only one solution by painting every square and *generate_hanjie_empty(+Width, +Height, -ColumnHints, -RowHints)* that generates a puzzle with only one solution by leaving every square blank.

4 Decision Variables

The first set of decision variables created to solve the described problem, are the elements of the puzzle grid which can either be instantiated to 0 or to 1, where 0 represents a blank square and 1 represents a painted square. Additionally, if the clues are also uninstantiated, they are given a domain ranging from 0 to the length of the row and the sum of the elements in the row and of the elements in the list of clues are also stored in a variable.

5 Constraints

For the implementation of a solution to the problem in *Prolog* with constraint logic programming, several constraints had to be created. To establish these constraints, the puzzle grid's rows are searched at the same time as the list of clues for the rows. For each set of rows and their corresponding clues, the first constraint added is that the sum of all their elements must have the same value since the number of painted squares, which correspond to 1s must be equal to the total of the clues, and every other square must be blank. This is done through the use of the predicate *sum/3*, using the operator `#=` as the second argument for the predicate and a variable for the third which will be the same for the sum of the row elements and of the clues. In addition to these constraints, an *automaton/3* constraint is created for each row to ensure the sequence of the colored blocks. This automaton will be an NFA and it starts by creating a loop between the initial state and itself by reading 0s. From the initial state it can move on to the next state by reading a 1. It must then receive a number of 1s equivalent to the remainder of the clue, which will be decremented until it reaches 0. When this happens, another loop with input 0 is created from the current state to itself along with an additional transition from this state to the next one, also by reading a 0. This ensures the painted blocks are separated by at least one blank square. This is repeated for every clue until the list is empty. However, this solution alone could fail if the last painted square was at the end of the row as there would be no blank squares after it and the automaton would fail to advance to the next, and final, state. To solve this, a 0 is appended to every row before the automaton constraint is established on this new list.

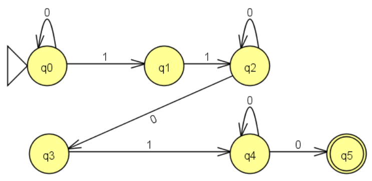


Fig. 2. An example of the automaton created for a row with the set of clues [2,1]

After these constraints for the rows are established, a list of the columns is obtained by using the predicate *transpose/2* of the *SICStus Prolog* 4.2.3 lists library on the puzzle grid and the process is repeated for this list and the list of corresponding clues.

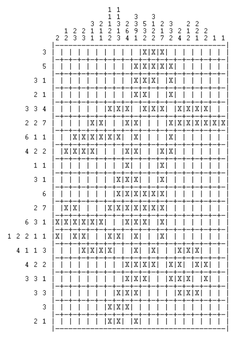
6 Search Strategy

For the adopted search strategy, the predicate *labeling/2* is called for a list of variables containing the elements of the puzzle grid, ordered as seen on the grid from left to right and top to bottom, with the option `ff` in the option list. It was chosen to use the *first fail* strategy to attempt to reduce the resulting search tree by failing as soon as possible and thus achieving a shorter execution time.

7 Solution Presentation

For printing the achieved solution in text mode, the predicate *draw_grid(+Grid, +ColumnHints, +RowHints)* was created. This predicate begins by obtaining the size of the largest sublist in *RowHints* when every element is printed with an additional space in front. This is done with the predicate *largest_sublist(+List, -Size)* and allows the printing predicates to know how much horizontal spacing they must apply when printing the solutions.
Afterwards, the column hints are drawn by first determining size of the largest sublist, allowing the predicate to know how many lines it will take to print all the hints, and printing each hint in the correct position. Finally, the grid is printed on the screen, along with the row hints, with 'X's representing painted squares and spaces representing blank squares. Between each column and each row, as well as on the borders of the grid, a separator is drawn to accentuate the grid structure.
Additionally, we allow the results to be saved to a file as sometimes, for very large puzzles, the display will not fit properly in the terminal. To do this, use the predicate *hanjie_solve_to_file(ColHints, RowHints, FilePath)* where **FilePath** must

lead to a file that the program has permission to write in, or create in case it doesn't exist.



```
                       1 1
                       1 1   3   3
                   3 2 1 3 2 3 5 1 2 3   2 2
             1 2 2 1 1 2 1 6 9 3 2 1 3 2 1 2 2
           2 2 3 3 1 1 2 3 4 1 2 2 7 2 4 2 1 2 1 1
          |---------------------------------------|
        3 | | | | | | | | | | |X|X|X| | | | | | | |
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
        5 | | | | | | | | | |X|X|X|X|X| | | | | | |
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
      3 1 | | | | | | | | | |X|X|X| |X| | | | | | |
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
      2 1 | | | | | | | | | |X|X| | |X| | | | | | |
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
    3 3 4 | | | | | | |X|X|X| |X|X|X| |X|X|X|X| | |
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
    2 2 7 | | | | |X|X| | |X|X| | | |X|X|X|X|X|X|X|
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
    6 1 1 | | |X|X|X|X|X|X| |X| | | |X| | | | | | |
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
    4 2 2 | |X|X|X|X| | | |X|X| | |X|X| | | | | | |
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
      1 1 | | | | | | | | |X| | | |X| | | | | | | |
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
      3 1 | | | | | | | |X|X|X| | |X| | | | | | | |
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
        6 | | | | | | | |X|X|X|X|X|X| | | | | | | |
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
      2 7 | |X|X| | | |X|X|X|X|X|X|X| | | | | | | |
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
    6 3 1 |X|X|X|X|X|X| | |X|X|X| |X| | | | | | | |
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
1 2 2 1 1 |X| |X|X| | |X|X| |X| | |X| | | | | | | |
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
  4 1 1 3 | | | |X|X|X|X| | |X| |X| | |X|X|X| | | |
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
    4 2 2 | | | | | | | | |X|X|X|X| |X|X| |X|X| | |
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
    3 3 1 | | | | | | | | |X|X|X| | |X|X|X| |X| | |
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
      3 3 | | | | | | | |X|X|X| | | | |X|X|X| | | |
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
        3 | | | | | | |X|X|X| | | | | | | | | | | |
          |-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-|
      2 1 | | | | | | |X|X| |X| | | | | | | | | | |
          |---------------------------------------|
```

Fig. 3. A solution to a *Hanjie* puzzle as viewed in text mode by the developed program.

8 Results

When testing our application with several examples of *hanjie* puzzles, we noticed it solved any puzzle with only one possible solution in under a second for puzzles with dimensions up to 80x80. Most of this time however was for preprocessing, as the labeling phase by itself takes less than 10 milliseconds for puzzles with dimensions up to 250x250. Concluding that there are no alternate solutions is also done in under a millisecond. For puzzles with dimensions greater than the previously indicated, the increase in execution time is linear relatively to the area of the puzzle grid.

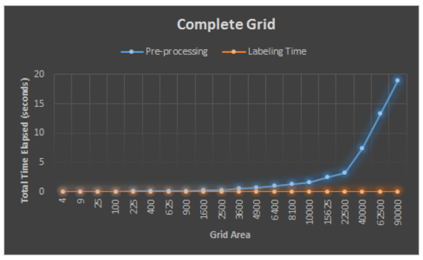


Fig. 4. Execution time for solving a grid where every square is painted. Automatons generated for each row/column will be large and therefore require a larger pre-processing duration.

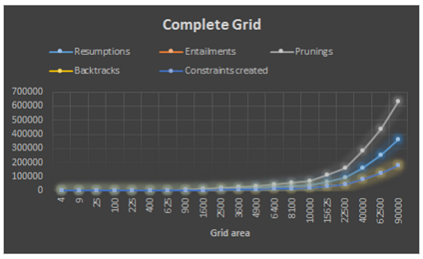


Fig. 5. Constraint programming specific statistics for solving a full grid. The adopted search strategy will lead to a large number of backtracks in this case as it tries smaller values first.

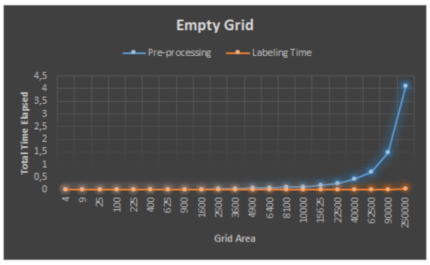


Fig. 6. Execution time for solving a grid where every square is blank. The chosen search strategy is ideal for this kind of puzzle as no backtracking will be required.

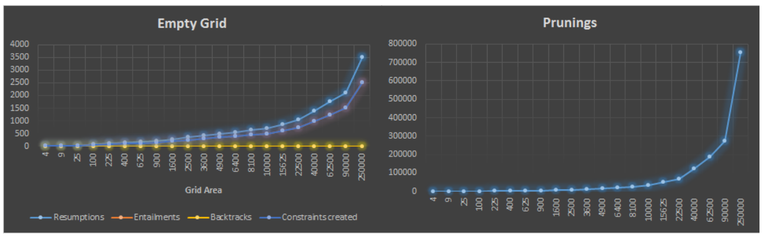


Fig. 7. Constraint programming specific statistics for solving an empty grid. The adopted search strategy will lead to no backtracks.

When the number of possible solutions increases however, the labeling time increases exponentially for finding the first solution. The labeling phase for other solutions that do not require the total number of painted squares in the row to be altered is processed in under a millisecond. Finding that there are no solutions left, or other solutions that do require a change in the total number of painted squares of a row, takes approximately the same time as finding the previous set of solutions.

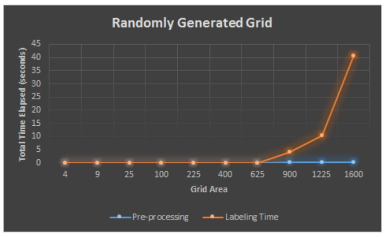


Fig. 8. Execution time for solving a randomly generated grid which can have multiple solutions.

These results were obtained using an Intel(R) Core(TM) i7-2630QM CPU @ 2.00GHz 2.00GHz with 6,00 GB RAM.

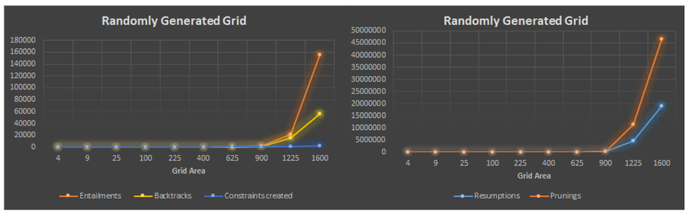


Fig. 9. Constraint programming specific statistics for solving a randomly generated grid which can have multiple solutions.

9 Conclusions and Future Work

This project allowed us to understand the mechanics behind constraint logic programming studied during Logic Programming classes, by applying them to a logic program capable of solving the 2D puzzle, *Hanjie*.
While trying to find a solution for this problem, the most complex part was developing the restrictions that would ensure the integrity and sequence of the painted squares. We quickly realized using strictly arithmetic restriction would create an overly complex solution and so decided to implement combinatorial constraints instead. More specifically, we implemented the automaton restriction since *Hanjie* is a puzzle of sequences in two dimensions.
As the results we obtained show, what makes these puzzles computationally heavy to solve, are the number of possible solutions, and not the dimensions of the puzzle alone. Although larger dimensions, do allow for more possible solutions, once restraints have been applied, if there is only one possible solution, the labeling phase is quickly processed. It is therefore possible to solve a puzzle with dimensions 100x100 a lot faster than a 30x30 puzzle if the restraints leave a smaller search base for that puzzle.
The implementation developed for this project will quickly solve any puzzles whose clues strongly restrict the elements of the rows/columns but will suffer an exponential growth in execution time with more ambiguous clues or puzzles with more than one solution. For puzzles with dimensions up to 30x30 however, it can usually solve them within 1-5 seconds. Additionally, it will always find a solution for a puzzle, provided it has any.
If we had more time to work on this project, we would spend it analyzing the logic behind *Hanjie*, to be able to deduce more restrictions from the sets of clues that could be applied to our decision variables and help shorten execution time by reducing the resulting search base for possible solutions.

10 Bibliography

References

1. Russel, Stuart; Norvig, Peter: Artificial Intelligence: A Modern Approach. Pearson, New Jersey (2009)
2. Sterling, Lehon; Shapiro, Ehud: The Art of Prolog: Advanced Programming Techniques. The MIT Press, Massachusetts (1994)
3. FEUP - Logic Programming class support documentation, `http://moodle.up.pt/course/view.php?id=511`
4. Description of Hanjie puzzles `http://www.puzzlemix.com/Hanjie`
5. Ullman, Jeffrey; Motwani, Rajeev; Hopcroft, John: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley (2001)
6. Description of automaton and other combinatorial constraints available in *SICStus Prolog* 4.2.3 `http://www.fi.muni.cz/~hanka/sicstus/doc/html/sicstus/Combinatorial-Constraints.html`

11 Annex A - Output to File Example

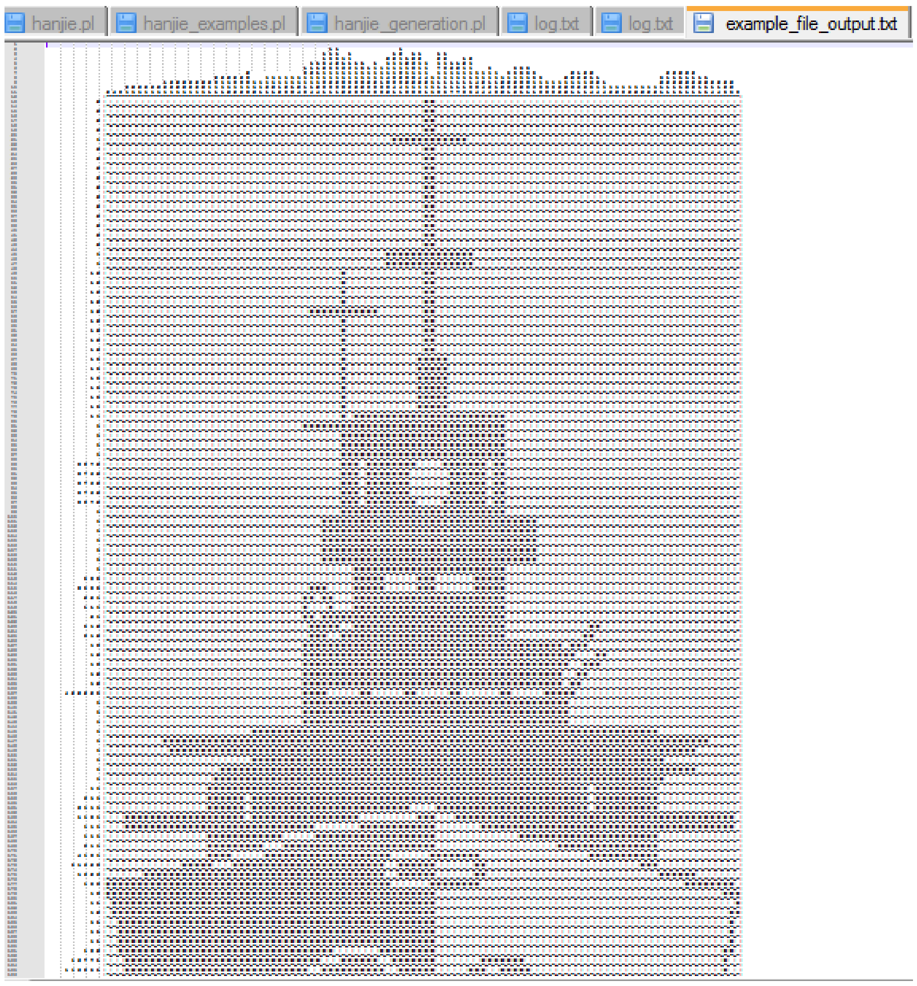


Fig. 10. The visualization of a text file created with this program.

12 Annex B - Code

'hanjie.pl':

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

?- ensure_loaded('hanjie_examples.pl').
?- ensure_loaded('hanjie_generation.pl').

%FOR EXAMPLES OF HANJIE PUZZLES, LOAD FILE "
    hanjie_examples.pl"

%predicate fo solving hanjie puzzles
%initializes a grid (list of lists) of uninstanciated
    variables with the apropriate size for the specified
    column and row hints.
%a hint must be a list of lists where every sublist
    contains the size and sequence of gray blocks in that
    line
%e.g.: [[1,1],[1]] means the first line must have two
    blocks of gray squares with length one and the second
    line must only have one
%the grid is then flattened to restrict the domain of the
     variables to [0,1].
%additional constrains are added with the predicate
    hanjie_constraints/3
%afterwards, the labeling/2 predicate is used and the
    resulting grid is drawn on screen
%IMPORTANT: EVERY ROW/COLUMN MUST HAVE A HINT, EVEN IF IT
     IS 0, OTHERWISE THE SIZE OF THE GRID WOULD HAVE TO BE
     INDICATED MANUALLY
hanjie(ColHints,RowHints):-
        length(ColHints,GridWidth),
        length(RowHints,GridHeight),
        make_grid(Solution,GridWidth,GridHeight),
        flatten(Solution,Vars),
        domain(Vars,0,1),
        !,
        hanjie_constraints(ColHints,RowHints,Solution),
        flatten(ColHints, TotalColHints),flatten(RowHints
            , TotalRowHints),
        sum(TotalColHints, #=, TotalValue), sum(
            TotalRowHints, #=, TotalValue),
        sum(Vars, #=, TotalValue),
        labeling([ff],Vars),
        draw_grid(Solution,ColHints,RowHints).

%this predicate allows the puzzle solution to be stored
    in a file given by the specified path
hanjie_solve_to_file(ColHints,RowHints, Filepath):-
        open(Filepath,append,S1),
        length(ColHints,GridWidth),
        length(RowHints,GridHeight),
        make_grid(Solution,GridWidth,GridHeight),
        flatten(Solution,Vars),
        domain(Vars,0,1),
        !,
        hanjie_constraints(ColHints,RowHints,Solution),
        flatten(ColHints, TotalColHints),flatten(RowHints
            , TotalRowHints),
        sum(TotalColHints, #=, TotalValue), sum(
            TotalRowHints, #=, TotalValue),
        sum(Vars, #=, TotalValue),
        labeling([ff],Vars),
        draw_grid(S1,Solution,ColHints,RowHints),
        close(S1).

%this predicate will generate a random hanjie puzzle and
    solve it by calling hanjie_generate and hanjie
hanjie_generate_and_solve(NumCols,NumRows):-
        generate_hanjie(NumCols,NumRows,ColHints,RowHints
            ),
        !,
        hanjie(ColHints,RowHints).

%does the same as hanjie/2 but measures statistics for
    calculating the solution instead of drawing the grid
hanjie_stats(ColHints,RowHints):-
        length(ColHints,GridWidth),
        length(RowHints,GridHeight),
        make_grid(Solution,GridWidth,GridHeight),
        flatten(Solution,Vars),
        domain(Vars,0,1),
        !,
        print('dumping previous fd stats:'),nl,
        fd_statistics,
        statistics(walltime,[W0|_]),
        statistics(runtime,[T0|_]),
        hanjie_constraints(ColHints,RowHints,Solution),
        flatten(ColHints, TotalColHints),flatten(RowHints
            , TotalRowHints),
        sum(TotalColHints, #=, TotalValue), sum(
            TotalRowHints, #=, TotalValue),
        sum(Vars, #=, TotalValue),
        statistics(walltime,[W1|_]),
        statistics(runtime,[T1|_]),
        labeling([ff],Vars),
        statistics(walltime,[W2|_]),
        statistics(runtime,[T2|_]),
        T is T1-T0,
        W is W1-W0,
        Tl is T2-T1,
        Wl is W2-W1,
        nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,
        format('creating constraints took CPU ~3d sec.~n
            ', [T]),
        format('creating constraints took a total of ~3d
            sec.~n', [W]),
        format('labeling took CPU ~3d sec.~n', [Tl]),
        format('labeling took a total of ~3d sec.~n', [Wl
            ]),
        nl,
        fd_statistics.

%creates the constrains for each row of the grid using
    restrict_rows/2 and the row hints.
%uses the predicate transpose/2 (lists library) to obtain
     a list of the columns and creates the constraints for
%them as well, using restrict_rows/2 and the column hints
hanjie_constraints(ColHints,RowHints,Grid):-
        restrict_rows(Grid,RowHints),
        transpose(Grid,Columns),
        restrict_rows(Columns,ColHints).

%restricts the domain of each element of the grid by
    ensuring the sum of all elements is equal to the sum
    of the hints
%(0 corresponds to blank squares and 1 corresponds to
    gray squares of the hanjie puzzle)
%to ensure the sequences of colored squares are
    maintained, this predicate creates an automaton for
    each row
restrict_rows([],[]).
restrict_rows([R|Rs],[H|Hs]):-
```

```
        length(R,MaxSum),
        (%This 'or' allows lists to be completely
            uninstantiated
        var(H),
        HintLength is floor((MaxSum+1)/2),
        length(H,HintLength),
        checkHints(H,MaxSum)
        ;
        nonvar(H),
        checkHints(H,MaxSum)
        ),
        RowSum #=< MaxSum,
        sum(H,#=,RowSum),
        sum(R,#=,RowSum),
        create_transitions(H, Arcs, start, FinalState,1),
        append(R, [0], RowWithExtraZero), %a zero is
            added to simplify the automaton (every gray
            block must be followed by at least one blank
            square, even the last one)
        automaton(RowWithExtraZero, [source(start), sink(
            FinalState)], [arc(start,0,start) | Arcs]),
        restrict_rows(Rs,Hs).

%checks if the hints are variables. If so, the domain is
    assigned to the variable
checkHints([],_):-!.
checkHints([H|Hs],MaxSum):-
        (var(H),domain([H],0,MaxSum);
        integer(H)),
        checkHints(Hs,MaxSum).

%this predicate is used to generate the transitions (arcs
    ) between each state of the automaton
%it goes through every value of the Hint list,
    decrementing them until they reach 0 and creating
    mandatory transitions
%to ensure continuous gray blocks. When the Hint reaches
    0 it creates transitions to ensure at least one blank
    block after the gray one
%(to allow for 1 or more white squares after each gray
    block, the automaton will be an NFA)
%if the first square of a block has a hint with value 0 (
    FirstSquare = 1), the CurState is set as NextState (
    hint is ignored)
create_transitions([], [], FinalState, FinalState,_).
create_transitions([Hint|Hs], Transitions, CurState,
    FinalState,FirstSquare) :-
     (Hint #= 0, %finished current 'gray' blocks segment
        %'gray' blocks must be followed by AT LEAST ONE '
            blank' square (loop to current state with '
            blank' blocks)
        %(an extra 'blank' square was added to end of the
             row for the case when the 'gray' block ends
            at the grid's border)
     (FirstSquare =:=0,
                 Transitions = [arc(CurState,0,CurState),
                     arc(CurState,0,NextState) | Ts],
                 create_transitions(Hs, Ts, NextState,
                     FinalState,1);
        FirstSquare =:=1,
                 Transitions = [arc(CurState,0,CurState)],
                 create_transitions(Hs, Ts, CurState,
                     FinalState,1))
        ;
        %in this case, we aren't finished with the gray
            block yet and as such need more 'gray' squares
             to advance
        Hint #> 0,
     Transitions = [arc(CurState,1,NextState) | Ts],
     NewHint #= Hint-1,
     create_transitions([NewHint|Hs], Ts, NextState,
        FinalState,0)).


%------------------------------------------------------------
    UTILITIES
    ------------------------------------------------------------

%flattens a list of lists into a single list
flatten([],[]):-!.
flatten([[]|Gs],Fg):-
        !,
        flatten(Gs,Fg).
flatten([[G1|G1s]|Gs],[G1|Fg]):-
        flatten([G1s|Gs],Fg).
```

```
%this predicate is used to create a grid (list of lists)
    of uninstanciated variables with the specified
    dimensions
make_grid(Grid,Width,Height):-
        length(Grid,Height),
        make_rows(Grid,Width).
make_rows([],_):-!.
make_rows([G|Gs],Width):-
        length(G,Width),
        make_rows(Gs,Width).

%this predicate is used to determine the size of the
    largest sublist in a list of lists
largest_sub_list([],0):-!.
largest_sub_list([L|Ls],N):-
        largest_sub_list(Ls,M1),
        length(L,M2),
        (M1 > M2,
        N is M1;
        N is M2),
        !.
%------------------------------------------------------------END
    UTILITIES
    ------------------------------------------------------------------


%----------------------------------------------------------DISPLAY
    PREDICATES
    ----------------------------------------------------------------

%predicates for drawing the hanjie's puzzle grid
sign(0,' ').
sign(1,'X').
draw_grid([B|Bs],ColHints,RowHints):-
        largest_sub_list(RowHints,HChars),
        HSpacing is HChars*2,
        nl,nl,
        HSpacingForCols is HSpacing+1,
        draw_column_hints(ColHints,HSpacingForCols),
        length(B,N),
        putChars(' ',HSpacing),
        print('|'),print_separator(N,'-'),
        pg([B|Bs],RowHints,HSpacing),
        putChars(' ',HSpacing),
        print('|'),print_separator(N,'-'),nl.
pg([B],[RH],HLength) :-
```

```
          draw_row_hints(RH,HLength),
          print_line(B),  !.
pg([B|Bs],[RH|RHs],HLength)  :-
          draw_row_hints(RH,HLength),
          print_line(B),
          length(B,N),
          putChars('  ',HLength),
          print('|'),
          print_separator(N,'+'),
          pg(Bs,RHs,HLength).
print_line([])  :-
          print('|'),nl.
print_line([L|Ls]):-
          sign(L,Symbol),
          print('|'),
          print(Symbol),
          print_line(Ls).
print_separator(1,_):-print('-|'),  nl.
print_separator(N,MidChar):-
          N > 1,
          N1 is N-1,
          print('-'),print(MidChar),
          print_separator(N1,MidChar).

putChars(_,0):-!.
putChars(Char,N):-
          N > 0,
          N1 is N-1,
          print(Char),
          putChars(Char,N1).

draw_column_hints(ColHints,HSpacing):-
          largest_sub_list(ColHints,VSpacing),
          dch(ColHints,HSpacing,VSpacing).
dch(_,_,0):-!.
dch(ColHints,HSpacing,VSpacing):-
          VSpacing > 0,
          putChars('  ',HSpacing),
          draw_elements_at_vpos(ColHints,  VSpacing),nl,
          NewVSpacing  is  VSpacing-1,
          dch(ColHints,HSpacing,NewVSpacing).
draw_elements_at_vpos([],_):-!.
draw_elements_at_vpos([CH|CHs],VSpacing):-
          length(CH,NumHints),
          (NumHints < VSpacing,!,
          print('   ')
          ;
          ElementPos is NumHints-VSpacing,
          nth0(ElementPos,CH,Value),
          (Value < 10,!,
           print(Value);
           print('#')),
          print(' ')),
          draw_elements_at_vpos(CHs,VSpacing).

draw_row_hints([],_):-!.
draw_row_hints([R|Rs],HSize):-
     HSize>0,
          length([R|Rs],L),
          (HSize > L*2,!,
          NewHSize is HSize-2,
          print('   '),
          draw_row_hints([R|Rs],NewHSize)
          ;
          NewHSize is HSize-2,
          (R<10,!,
          print(R),print(' ');
          print('# ')),
          draw_row_hints(Rs,NewHSize)).
%-------------------------------------------------END DISPLAY
    PREDICATES
    ----------------------------------------------------------

%-----------------------------------------------------FILE I/O
    PREDICATES
    ----------------------------------------------------------
%predicates for drawing the hanjie's puzzle grid in a
    file
draw_grid(Stream,[B|Bs],ColHints,RowHints):-
          largest_sub_list(RowHints,HChars),
          HSpacing is HChars*2,
          write(Stream, '\n\n'),
          HSpacingForCols is HSpacing+1,
          draw_column_hints(Stream, ColHints,
              HSpacingForCols),
          length(B,N),
          putChars(Stream,'  ',HSpacing),
          write(Stream,'|'),print_separator(Stream,N,'-'),
          pg(Stream,[B|Bs],RowHints,HSpacing),
          putChars(Stream,'  ',HSpacing),
          write(Stream,'|'),print_separator(Stream,N,'-'),
              write(Stream,'\n').
pg(Stream,[B],[RH],HLength) :-
          draw_row_hints(Stream,RH,HLength),
          print_line(Stream,B), !.
pg(Stream,[B|Bs],[RH|RHs],HLength) :-
          draw_row_hints(Stream,RH,HLength),
          print_line(Stream,B),
          length(B,N),
          putChars(Stream,'  ',HLength),
          write(Stream,'|'),
          print_separator(Stream,N,'+'),
          pg(Stream,Bs,RHs,HLength).
print_line(Stream,[]) :-
          write(Stream,'|\n').
print_line(Stream,[L|Ls]):-
          sign(L,Symbol),
          write(Stream,'|'),
          write(Stream,Symbol),
          print_line(Stream,Ls).
print_separator(Stream,1,_):-write(Stream,'-|'), write(
    Stream,'\n').
print_separator(Stream,N,MidChar):-
          N > 1,
          N1 is N-1,
          write(Stream,'-'),write(Stream,MidChar),
          print_separator(Stream,N1,MidChar).

putChars(_,_,0):-!.
putChars(Stream,Char,N):-
          N > 0,
          N1 is N-1,
          write(Stream,Char),
          putChars(Stream,Char,N1).

draw_column_hints(Stream,ColHints,HSpacing):-
          largest_sub_list(ColHints,VSpacing),
          dch(Stream,ColHints,HSpacing,VSpacing).
dch(_,_,_,0):-!.
dch(Stream,ColHints,HSpacing,VSpacing):-
          VSpacing > 0,
          putChars(Stream,'  ',HSpacing),
          draw_elements_at_vpos(Stream,ColHints, VSpacing),
              write(Stream,'\n'),
          NewVSpacing is VSpacing-1,
          dch(Stream,ColHints,HSpacing,NewVSpacing).
draw_elements_at_vpos(_,[],_):-!.
draw_elements_at_vpos(Stream,[CH|CHs],VSpacing):-
          length(CH,NumHints),
          (NumHints < VSpacing,!,
          write(Stream,'   ')
          ;
          ElementPos is NumHints-VSpacing,
          nth0(ElementPos,CH,Value),
          (Value < 10,!,
           write(Stream,Value);
           write(Stream,'#')),
          write(Stream,' ')),
          draw_elements_at_vpos(Stream,CHs,VSpacing).

draw_row_hints(_,[],_):-!.
draw_row_hints(Stream,[R|Rs],HSize):-
    HSize>0,
          length([R|Rs],L),
          (HSize > L*2,!,
          NewHSize is HSize-2,
          write(Stream,'   '),
          draw_row_hints(Stream,[R|Rs],NewHSize)
          ;
          NewHSize is HSize-2,
          (R<10,!,
          write(Stream,R),write(Stream,' ');
          write(Stream,'# ')),
          draw_row_hints(Stream,Rs,NewHSize)).
%------------------------------------------------------END DISPLAY
   PREDICATES
   ------------------------------------------------------------------
```

'hanjie_examples.pl':

```
example1([[1], [1]], [[2], [0]]).

example1_1([[1], [1, 1]], [[2], [0], [1]]).

example2([[1], [0], [1]], [[1], [0], [1]]). % this one
    has two solutions

example3([[2], [2, 1], [1, 1], [3], [1, 1], [1, 1],
     [2], [1, 1], [1, 2], [2]],
          [[2, 1], [2, 1, 3], [7], [1, 3], [2, 1]]).

example4([[2], [1, 2], [2, 3], [2, 3], [3, 1, 1], [2, 1,
    1], [1, 1, 1, 2, 2], [1, 1, 3, 1, 3], [2, 6, 4], [3,
    3, 9, 1], [5, 3, 2], [3, 1, 2, 2], [2, 1, 7], [3, 3,
    2], [2, 4], [2, 1, 2], [2, 2, 1], [2, 2], [1], [1]],
                   [[3], [5], [3, 1], [2, 1], [3, 3, 4],
                        [2, 2, 7], [6, 1, 1], [4, 2, 2], [1,
                         1], [3, 1], [6], [2, 7], [6, 3, 1],
                         [1, 2, 2, 1, 1], [4, 1, 1, 3], [4,
                        2, 2], [3, 3, 1], [3, 3], [3], [2,
                        1]]).

example5([[2, 3, 1, 3], [2, 1, 6, 1, 4], [2, 3, 3, 1, 3,
    2], [1, 8, 2, 4], [3, 4, 3, 2], [1, 9, 2, 1, 1], [4,
    13, 8], [2, 11, 9], [3, 1, 3, 1, 1, 1, 9], [1, 5, 1,
    5], [4, 3, 2, 3], [4, 4, 2, 1], [3, 2, 3, 5, 1], [1,
    3, 5, 4, 3], [4, 2, 4, 6], [4, 2, 3, 6], [4, 2, 4, 3,
    2], [1, 1, 1, 3], [2, 5, 2], [1, 5, 3, 2], [10, 5, 1],
     [4, 7, 6, 7], [4, 6, 5, 7], [2, 4, 1, 1, 6, 7], [3,
    1, 1, 1, 3, 3, 7], [3, 5, 1, 1, 2, 6], [10, 3, 1], [4,
     5, 1], [4, 3, 1, 1, 1], [4, 3, 2, 3, 3]],
                   [[9, 4, 9], [3, 1, 3, 3, 9], [1, 1, 1,
                        3, 2, 6], [2, 2, 3, 4], [3, 1, 1,
                        3], [7, 4, 5, 3], [6, 5, 8], [6, 5,
                        4, 5], [1, 4, 3, 10], [1, 1, 4, 5,
                        4], [1, 5, 6, 1], [3, 3, 1, 1, 3,
                        1], [3, 8, 2, 4], [3, 2, 9, 1, 1, 1,
                         1], [2, 11, 1, 2], [1, 3, 1, 1, 7],
                         [2, 2, 2, 6, 2], [1, 1, 12, 1], [1,
                         5, 1, 7], [1, 3, 3, 1, 1, 6], [5,
                        5], [7], [1, 1, 3, 6, 1, 1], [2, 4,
                        13, 1, 1], [3, 5, 6, 5, 1], [7, 2,
                             5], [1, 1, 5, 3, 5], [2, 1, 3, 1, 3,
                              5, 1], [5, 5, 1], [5, 4, 2]]).

%100x100
warship([[2], [4], [8], [2, 13], [2, 17], [2, 18], [2,
    19], [2, 19], [2, 19], [1, 2, 19], [2, 2, 19], [2, 2,
    19], [2, 2, 19], [2, 2, 19], [2, 2, 19], [2, 2, 18],
    [2, 26, 1], [2, 9, 17, 1], [2, 10, 17, 1], [2, 10, 16,
     2], [2, 9, 17, 2], [2, 9, 17, 2], [2, 3, 3, 16, 3],
    [13, 16, 2], [13, 15, 1], [12, 16, 2, 1], [12, 15, 2,
    1], [12, 15, 1, 2], [11, 15, 2, 3], [11, 15, 1, 3],
    [11, 14, 2, 4], [1, 3, 20, 14, 1, 3], [1, 1, 2, 23,
    13, 1, 4], [1, 1, 1, 21, 14, 2, 4], [1, 1, 5, 2, 22,
    13, 3, 3, 1], [1, 1, 5, 4, 5, 13, 13, 2, 3, 1], [1, 1,
     5, 2, 5, 13, 12, 2, 3, 1], [31, 1, 6, 13, 12, 3, 3,
    2], [1, 15, 1, 6, 13, 12, 3, 2, 2], [1, 29, 13, 12, 3,
     2, 3], [1, 5, 32, 13, 3, 2, 3], [1, 42, 13, 2, 3, 3],
     [1, 29, 12, 13, 2, 3, 3], [29, 12, 14, 1, 3, 3], [2,
    17, 10, 12, 4, 8, 3, 2], [1, 2, 17, 10, 12, 3, 8, 4,
    1], [1, 2, 17, 10, 12, 3, 2, 9, 4, 1], [1, 2, 17, 23,
    3, 2, 10, 3], [1, 2, 6, 8, 22, 3, 2, 11, 3], [1, 2,
    11, 19, 11, 4, 2, 11, 3], [38, 19, 11, 4, 2, 13, 1],
    [38, 19, 11, 25], [1, 2, 11, 7, 10, 12, 1, 1, 3], [1,
    2, 12, 8, 10, 12, 1, 1, 5], [1, 2, 17, 23, 1, 1, 7],
    [1, 2, 17, 24, 1, 1, 7], [1, 2, 17, 10, 13, 1, 1, 8],
    [2, 17, 10, 13, 1, 1, 3, 5], [29, 13, 4, 3, 4], [29,
    14, 2, 4, 4], [5, 19, 14, 1, 2, 4], [29, 14, 1, 1, 3],
     [44, 2, 2, 2], [5, 20, 2, 3, 1], [5, 5, 14, 2, 2, 2],
     [5, 5, 15, 3, 2, 1], [5, 5, 15, 3, 3, 1], [5, 5, 15,
    6, 1], [5, 15, 3, 3], [21, 3, 3], [21, 3, 3], [22, 2,
    2], [2, 17, 2, 2], [2, 2, 13, 3, 1], [1, 2, 13, 2, 1],
     [2, 2, 13, 2, 1], [2, 1 , 6, 5, 4], [1, 2, 14, 4],
    [1, 14, 4], [14, 2], [14, 2], [14, 3], [14, 2], [14,
    2], [15, 2], [15, 2], [15, 2], [5, 2, 1, 1], [2, 1, 2,
     1, 1], [2, 1, 2, 1, 1], [2, 1, 2, 1, 1], [2, 1, 2, 2,
     1], [2, 1, 2, 2, 1], [1, 2, 1, 1], [1, 2, 1, 1], [2,
    1, 4], [2, 1, 4], [2, 2, 5], [2, 2, 5], [3]],
                  [[2], [2], [2], [2], [12], [2], [2], [2],
                      [2], [2], [2], [2], [2], [2], [2],
                     [2], [14], [14], [1, 2], [1, 2], [1,
                     2], [1, 2], [11, 2], [1, 2], [1, 2],
                     [1, 2], [1, 2], [1, 5], [1, 5], [1,
                     5], [1, 5], [1, 5], [1, 5], [1, 24],
                     [32], [26], [26], [26], [3, 8, 7, 2],
                     [3, 7, 6, 2], [3, 7, 6, 2], [3, 7, 6,
                    2], [3, 8, 7, 2], [26], [34], [34],
                    [34], [34], [34], [24], [5, 3, 5], [3,
                     5, 3, 5], [2, 2, 24], [1, 1, 24], [2,
                     29], [5, 24, 2], [3, 26, 2], [41, 3],
                     [43, 2], [42, 3], [41, 2], [41, 2],
                    [4, 2, 2, 2, 2, 5], [42], [42], [42],
                    [64], [86], [83], [65], [75], [71],
                    [60, 10], [6, 53, 10], [6, 25, 24,
                    10], [19, 17, 3, 44], [30, 12, 40],
                    [12, 19, 22], [9, 24, 16], [4, 20, 8,
                    11], [5, 24, 6, 2, 3], [38, 6, 2, 6],
                    [45, 8, 8], [52, 3], [51, 1], [51, 2],
                     [50, 1], [50, 1], [50, 1], [34, 9,
                    2], [31, 6, 7, 7, 1], [29, 10, 6, 3,
                    5, 1], [27, 8, 5, 5, 3, 1], [25, 5, 5,
                     4, 10, 5, 2], [23, 3, 11, 2, 3, 3, 5,
                     1], [20, 3, 16, 2, 8, 11, 1], [18, 2,
                     8, 4, 1, 9, 3, 7, 2], [15, 4, 8, 5,
                    4, 12, 5, 9, 1], [11, 5, 7, 8, 2, 14,
                    5, 11, 1], [7, 7, 6, 13, 9, 15, 15]]).
%the following examples allow for multiple solutions (
    clues to be determined)
example_i_1([_, _],
                        [_, _]).
example_i_2([_, _, _],
                        [_, _, _]).

example_i_3([[1, 1], [_], [1, 1]], [[1, 1], [_], [1, 1]])
    .

%large amounts of completely uninstantiated lists (number
     of hints per row will be maximum) will possibly take
    long to compute
example_i_4([_, _, _, _, _, _, _, _, _, _, _, _, _, _, _
    ],
                        [_, _, _, _, _, _, _, _, _, _, _
                            , _, _, _, _]).
example_i_5([[_], [_], [_], [_], [_], [_], [_], [_], [_],
     [_], [_], [_], [_], [_], [_]],
                        [[_], [_], [_], [_], [_], [_], [
                            _], [_], [_], [_], [_], [_],
                            [_], [_], [_]]).
```

'hanjie_generation.pl':

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

?-ensure_loaded('hanjie.pl').


%this predicate will create a random hanjie grid and
    obtain the clues for generated solutions
generate_hanjie(Width,Height,ColHints,RowHints):-
        make_atom_grid(PuzzleGrid,Width,Height),
        generate_rows_hints(PuzzleGrid,RH),
        transpose(PuzzleGrid,Cols),
        generate_rows_hints(Cols,CH),
        strip_zeros(RH,RowHints),
        strip_zeros(CH,ColHints),
        nl,nl,nl,
        print('This is a possible solution (the larger
            the grid the more likely it will have multiple
             solutions):'),nl,
        draw_grid(PuzzleGrid,ColHints,RowHints).

%this predicate will create a hanjie grid with all
    squares painted
generate_hanjie_full(Width,Height,ColHints,RowHints):-
        make_atom_grid_full(PuzzleGrid,Width,Height),
        generate_rows_hints(PuzzleGrid,RH),
        transpose(PuzzleGrid,Cols),
        generate_rows_hints(Cols,CH),
        strip_zeros(RH,RowHints),
        strip_zeros(CH,ColHints).

%this predicate will create a hanjie grid with all
    squares blank
generate_hanjie_empty(Width,Height,ColHints,RowHints):-
        make_atom_grid_empty(PuzzleGrid,Width,Height),
        generate_rows_hints(PuzzleGrid,RH),
        transpose(PuzzleGrid,Cols),
        generate_rows_hints(Cols,CH),
        strip_zeros(RH,RowHints),
        strip_zeros(CH,ColHints).

strip_zeros([],[]):-!.
strip_zeros([Row|Rest],[Result|Rs]):-
        delete(Row,0,Temp),
        (Temp = [] ,
        Result = [0]
        ;
        Result=Temp) ,! ,
        strip_zeros(Rest,Rs).

generate_rows_hints([],[]):-!.
generate_rows_hints([Row|Rest],[RH|RHs]):-
        generate_hints_for_row(Row,RH),
        generate_rows_hints(Rest,RHs).

generate_hints_for_row([],[0]):-!.
generate_hints_for_row([0|Rest],[0|RHs]):-
        generate_hints_for_row(Rest,RHs).
generate_hints_for_row([1|Rest],[Head|RHs]):-
        generate_hints_for_row(Rest,[NextHead|RHs]),
        Head is NextHead+1.


make_atom_grid_empty([],_,0):-!.
make_atom_grid_empty([Row|Rest],Width,Height):-
        Height > 0,
        make_atom_row_empty(Row,Width),
        RemainingHeight is Height-1,
        make_atom_grid_empty(Rest,Width,RemainingHeight).
make_atom_row_empty([],0):-!.
make_atom_row_empty([0|Rs],Width):-
        Width > 0,
        RemainingWidth is Width-1,
        make_atom_row_empty(Rs,RemainingWidth).

make_atom_grid_full([],_,0):-!.
make_atom_grid_full([Row|Rest],Width,Height):-
        Height > 0,
        make_atom_row_full(Row,Width),
        RemainingHeight is Height-1,
        make_atom_grid_full(Rest,Width,RemainingHeight).

make_atom_row_full([],0):-!.
make_atom_row_full([1|Rs],Width):-
        Width > 0,
        RemainingWidth is Width-1,
        make_atom_row_full(Rs,RemainingWidth).

make_atom_grid([],_,0):-!.
make_atom_grid([Row|Rest],Width,Height):-
        Height > 0,
        make_atom_row(Row,Width),
        RemainingHeight is Height-1,
        make_atom_grid(Rest,Width,RemainingHeight).

make_atom_row([],0):-!.
make_atom_row([R|Rs],Width):-
        Width > 0,
        random(0,2,R),
        RemainingWidth is Width-1,
        make_atom_row(Rs,RemainingWidth).
```