

# Declarative Programming Report.

## Sudoku Hidato

Dayrene Fundora [dayfundoraglez@gmail.com](mailto:dayfundoraglez@gmail.com)

Faculty of Mathematics and Computation (MATCOM),  
University of Havana (UH), Cuba.

### 1. Preliminaries. Randomness

Randomness was achieved through the `System.Random[1]` module and the main functions used were `random` to obtain an arbitrary value, `randomR` to generate random numbers in a range, and `randoms` to construct an infinite sequence of random values. Such an infinite sequence is used by almost all the functions of the implementation from indexes that indicate which value is to be used and it has been essential since `random` and `randomR` (Figura2) both receive a `RandomGen` type object (a *random* generator) which can be built manually with the `mkStdGen` function from a “seed” value where, due to Haskell’s referential transparency, it must vary so that `mkStdGen` and therefore, `random` and `randomR` return different values in each call.

### 2. Representation

The Hidatos have been represented as a triple  $(M, I, F)$  where  $I$  is the position where the minimum value of the board is located,  $F$  the position of the maximum value and  $M$  a matrix of  $n$  rows and  $m$  columns where:

- the boxes that must be filled in to complete the sudoku have a value of  $-2$
- the boxes with a predefined value have an arbitrary number between 1 and  $k$ , where it is said that  $k$  is *the size of hidato* <sup>1</sup>
- there are “fictitious” boxes that serve to simulate the shape of the Hidato and have a value of *negative1*.

Figure 3 shows an example of a Hidato with this representation.

### 3. Generating Hidatos

Essentially, the procedure that is carried out is to generate a solution and from it choose the boxes that will be left with preset values.

---

<sup>1</sup> The value 1 has been kept as the minimum to generate all the hidato and the maximum value coincides with its size.

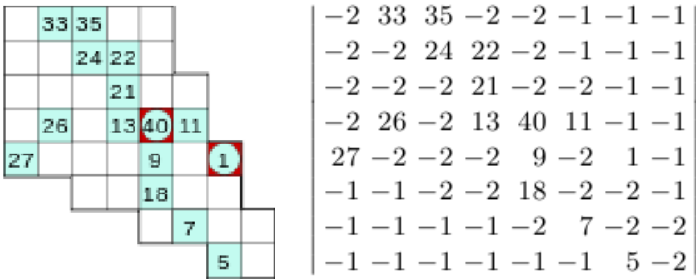
```
-- Aleatoriedad
lista = randoms (mkStdGen 115545) :: [Int]
```

**Figura 1.** “Global” sequence of *random* values in the implementation. The value `mkStdGen` receives to build it can be changed manually by any other.

```
random :: (RandomGen g, Random a) => g -> (a, g)

randomR :: (RandomGen g, Random a) :: (a, a) -> g -> (a, g)
```

**Figura 2.** Signature of `random` and `randomR`. Taken from [1, p.191, p.194]



**Figura 3.** Sudoku Hidato size 40 on the left and its matrix representation on the right.

### 3.1. Create a Solution

Given that a Hidato could also be represented as a sequence of cells that describe the “path” formed by the sequence of numbers that begins at 1 and ends at the size of the Hidato, the solutions are constructed using a recursive idea: Given the current position that it occupies, what is the next step to take considering the 8 options <sup>2</sup> available? This choice is made at random each time over the set of neighboring cells:

- to which no value has been assigned,
- in the case that they are outside the limits of the current matrix, the new matrix will have one more row and/or column, so the selected position is not an outside step (Figure 5).

```

movs :: [(Int,Int)]
movs = [(1,1), (1,-1), (1,0), (-1,-1), (-1,1), (-1,0), (0,1), (0,-1)]

```

**Figure 4.** The 8 movement options from a tile.

$$\begin{array}{c}
 \left| \begin{array}{c} 1 \end{array} \right| \\
 \left| \begin{array}{cc} 1 & 2 \end{array} \right| \\
 \left| \begin{array}{ccc} 1 & 2 & -1 \\ -1 & -1 & 3 \end{array} \right|
 \end{array}$$

**Figure 5.** Increase of the matrix according to the steps taken. When placing 2 on the board, the new matrix contains a column more to the right and when placing 3, a new column appears on the right and a row more.

Only one outer tile is chosen if there is no neighbor that is empty to reinforce the “density” of the sudoku game. We start from the matrix that has a single row and a column and it is considered a stop case when the value to be placed is the size that the Hydato is expected to have.

### 3.2. Choose Default Values

Once a solution is built, it is determined which cells will be empty and which cells will have default values. When developing the implementation, several ideas came up:

---

<sup>2</sup> Left, right, up, down and the 4 diagonal movements (Figure 4).

- Choose boxes randomly to prefix: in this case the Hidato is not guaranteed to have a unique solution.
- Choose which values of the board will be prefixed: specifically each time a number is arbitrarily chosen, its value and the value of the successor of its successor are prefixed on the board, thus restricting the number of places that the number in the center can occupy. Although this strategy is less fast than the first (since when determining the value to be prefixed, it requires detecting what position it occupies in the built solution) and it does not guarantee that the Hidato solution is unique, if it solves at least by reducing the amount of solutions.

These ideas are materialized with the functions `select_no_empty` and `select_better_no_empty` (Figure 7) respectively. 35% of the cells are filled, without considering those that contain the minimum and maximum value of the board, which will always be preset. It may be that increasing this percentage will guarantee the uniqueness of the solution but reduce the complexity of the “game”.

```
create_hidato :: Int -> Int -> ([[Int]], (Int,Int), (Int,Int), Int, [[Int]])
create_hidato len ri =
    let
        (hidato, init, final, ri') = create_hidato_aux len ([[ ]], (0,0)) (0,0) 1 ri
        (hidato', ri'') = select_better_no_empty (hidato, init, final, ri')
    in (hidato', init, final, ri'', hidato)
```

**Figura 6.** Function by which a Hidato is created. The parameter `len` is the size of the Hidato to be built and `ri` is the index of the list of random values from which to start working.

## 4. Solving Hidatos

To know the number of solutions of a given Hidato, the function that detects them exploits all the possibilities exhaustively (Figure 8). The idea is essentially recursive and defines that, given a position on the board, the solutions of the same will be those obtained from placing the next value of the path in each of the boxes that can be placed from the actual position. We start from the position where the minimum value of the board is located and the base cases are those in which the next value of the path is the maximum number and it is in a neighboring position to your location.

Of course, the procedure can be modified so that the recursion ends when a solution is found, but it was proceeded in this way to evaluate the strategy of prefixing the Hidato values. Either way, the problem of solving a Hidato can be reduced to the Boolean Satisfaction Problem, which is considered an NP-complete problem [2].

```

select_better_no_empty :: ([[Int]],(Int,Int),(Int,Int), Int) -> ([[Int]], Int)
select_better_no_empty (hidato, ip, (fr,fc), ri) =
    let
        t = (hidato !! fr) !! fc
        to_fill = div (35*t) 100
        (n,m) = shape hidato
        shidato = [[val | y <- [0..m-1], let val = selectv hidato (x,y) ip (fr
            to_select = not_settedfunc shidato t
    in select_better_no_empty_aux hidato shidato to_fill ri to_select

select_better_no_empty_aux :: [[Int]] -> [[Int]] -> Int -> Int -> [Int] -> ([[Int]], Int)
select_better_no_empty_aux complete_hidato shidato to_fill ri to_select
    | to_fill < 1 = (shidato, ri)
    | otherwise =

```

**Figura 7.** Segments of the functions by which the default values are determined.

## 5. Start the “Game”

Using the function `hidatos_list` with signature `hidatos_list :: Int ->Int ->Int ->String (hidatos_list n len ri)` it is specified that you want to generate and solve  $n$  Hidatos of size  $len$ . By functions defined in `Utils.hs` the Hidato and all its solutions are printed. The value  $ri$  in this call must be assigned the value 0 as shown in Figure 9 to start using the sequence of random numbers from the beginning.

## Referencias

1. Lipovaca, Miran: Learn you a Haskell for a great good!, Cap9, Randomness, 2011.
2. Bartoš, Samuel: Effective encoding of the Hidato and Numbrix puzzles to their CNF representation, Univerzita Karlova, Matematicko-fyzikální fakulta, 2014

```

solver_hidato_aux :: Int -> Int -> [[Int]] -> (Int,Int) -> [[[Int]]]
solver_hidato_aux len value hidato actual_pos solutions setted
    | value == (len-1) && near_final hidato actual_pos len = [hidato]
    | otherwise =
        let
            solutions' = [solver_hidato_aux len (value+1) hidato'
                          in myconcat solutions']

solver_hidato :: ([[Int]], (Int,Int), (Int,Int)) -> [[[Int]]]
solver_hidato (hidato, init, (fr,fc)) =
    let
        len = ((hidato !! fr) !! fc)
    in solver_hidato_aux len 1 hidato init [] (settedfunc hidato)

```

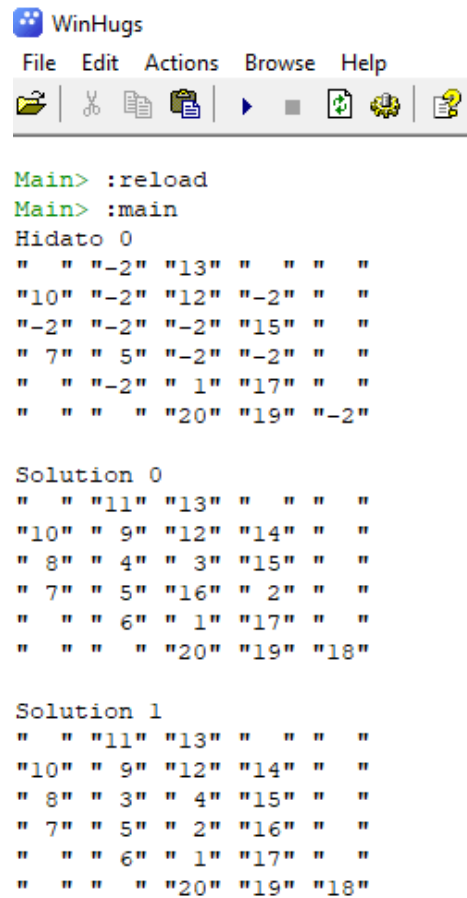
**Figura 8.** Segments of the functions through which the Hidato is solved.

```

main =
    do
        putStr (hidatos_list 5 20 0)

```

**Figura 9.** Example of invoking the “game” in textttMain.hs



```
WinHugs
File Edit Actions Browse Help
Main> :reload
Main> :main
Hidato 0
" " "-2" "13" " " " " "
"10" "-2" "12" "-2" " " "
"-2" "-2" "-2" "15" " " "
" 7" " 5" "-2" "-2" " " "
" " "-2" " 1" "17" " " "
" " " " "20" "19" "-2"

Solution 0
" " "11" "13" " " " " "
"10" " 9" "12" "14" " " "
" 8" " 4" " 3" "15" " " "
" 7" " 5" "16" " 2" " " "
" " " 6" " 1" "17" " " "
" " " " "20" "19" "18"

Solution 1
" " "11" "13" " " " " "
"10" " 9" "12" "14" " " "
" 8" " 3" " 4" "15" " " "
" 7" " 5" " 2" "16" " " "
" " " 6" " 1" "17" " " "
" " " " "20" "19" "18"
```

**Figura 10.** Section of the output of the “Game ” with the invocation of Figure 9.