

Sudoku Hidato

Masiel Villalba Carmenate villalbamasiel@gmail.com

Facultad de Matemática y Computación (MATCOM),
Universidad de la Habana (UH), Cuba.

1. Preliminares. Aleatoriedad

La aleatoriedad se consiguió mediante el módulo `System.Random[1]` y las principales funciones empleadas fueron `random` para obtener un valor arbitrario, `randomR` para generar números aleatorios en un rango y `randoms` para construir una secuencia infinita de valores aleatorios. Tal secuencia infinita es utilizada por casi todas las funciones de la implementación a partir de índices que indican cuál es el valor que les corresponde utilizar y ha sido primordial puesto que `random` y `randomR` (Figura2) reciben ambos un objeto tipo `RandomGen` (un generador *random*) que puede construirse manualmente con la función `mkStdGen` a partir de un valor “semilla” donde, con motivo de la transparencia referencial de Haskell, debe de variar para que `mkStdGen` y por consiguiente, `random` y `randomR` devuelvan valores diferentes en cada llamado.

2. Representación

Los Hidatos han sido representados como una tripla (M, I, F) donde I es la posición donde está ubicado el valor mínimo del tablero, F la posición del valor máximo y M una matriz de n filas y m columnas donde:

- las casillas que deben de rellenarse para completar el sudoku tienen asignado valor -2,
- las casillas con valor predefinido tienen un número arbitrario entre 1 y k , donde se dice que k es el *tamaño del hidato*¹,
- existen casillas “ficticias” que sirven para simular la forma del Hidato y tienen valor -1.

La Figura 3 muestra un ejemplo de un Hidato con esta representación.

3. Generando Hidatos

Esencialmente, el procedimiento que se realiza es generar una solución y a partir de ella elegir las casillas que se quedarán con valores prefijados.

¹ Se a mantenido el valor 1 como el mínimo para generar todos los hidatos y el valor máximo coincide con el tamaño del mismo.

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

Figura 1. Secuencia “global” de valores *random* en la implementación. El valor que recibe *mkStdGen* para construirla puede ser cambiado manualmente por cualquier otro.

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

Figura 2. Signatura de *random* y *randomR*. Tomado de [1, p.191, p.194]

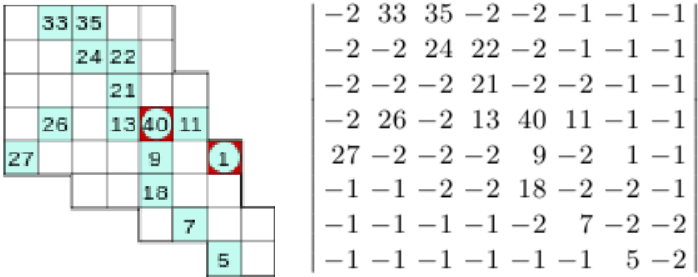


Figura 3. Sudoku Hidato de tamaño 40 a la izquierda y su representación matricial a la derecha.

3.1. Crear una solución

Dado que un Hidato pudiera representarse también como una secuencia de casillas que describen el “camino” que forma la secuencia de números que comienza en 1 y termina en el tamaño del Hidato, las soluciones se construyen mediante una idea recursiva: dada la posición actual que se ocupa, cuál es el siguiente paso que se va a dar considerando las 8 opciones² con las que se cuenta? Esta elección se toma cada vez de manera aleatoria sobre el conjunto de las celdas vecinas:

- a las cuales no se les ha asignado ningún valor,
- sobre aquellas que se salen de los límites de la matriz actual. En este caso la nueva matriz que representa al Hidato tendrá una fila y/o columna más respecto a la anterior, de forma tal que en ella la posición seleccionada no sea un paso exterior (Figura 5).

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

Figura 4. Las 8 opciones de movimiento desde una casilla.

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

Figura 5. Aumento de la matriz según los pasos tomados. Al ubicar al 2 en el tablero la nueva matriz contiene una columna más a la derecha y al ubicar al 3 aparece una nueva columna a la derecha y una fila más.

Se elige únicamente una casilla exterior si no existe ninguna vecina que esté vacía para reforzar la “densidad” del sudoku. Se parte de la matriz que tiene una sola fila y una columna y se considera caso de parada cuando el valor que corresponde colocar es el del tamaño que se ha previsto que tenga el Hidato.

² Izquierda, derecha, arriba, abajo y los 4 movimientos en diagonal (Figura 4).

3.2. Elegir valores predeterminados

Una vez construida una solución, se determina qué celdas estarán vacías y cuáles tendrán valores predeterminados. Al desarrollar la implementación, surgieron varias ideas:

- Elegir casillas aleatoriamente para prefijar: en este caso no se garantiza que el Hidato tenga solución única.
- Elegir qué valores del tablero se prefijarán: concretamente cada vez que se elige arbitrariamente un número, se prefijan en el tablero su valor y el del sucesor de su sucesor, así se restringe la cantidad de lugares que puede ocupar el número del centro. Aunque esta estrategia es menos rápida que la primera (puesto que al determinar el valor a prefijar requiere detectar qué posición ocupa en la solución construida) y tampoco garantiza que la solución del Hidato sea única, sí resuelve al menos que se reduzca la cantidad de soluciones.

Estas ideas se materializan con las funciones `select_no_empty` y `select_better_no_empty` (Figura 7) respectivamente. Se rellenan el 35% de las celdas, sin considerar las que contienen el valor mínimo y máximo del tablero que siempre serán prefijadas. Puede ser que aumentar este porcentaje consiga garantizar la unicidad de la solución pero reduciría la complejidad del “juego”.

```
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. Función mediante la cual se crea un Hidato. El parámetro `len` es el tamaño del Hidato que se desea construir y `ri` es el índice de la lista de valores aleatorios a partir del cual se comienza a trabajar.

4. Solucionando Hidatos

Para conocer la cantidad de soluciones de un Hidato dado, la función que las detecta explora todas las posibilidades de manera exhaustiva (Figura 8). La idea en esencia es recursiva y define que, dada una posición del tablero, las soluciones del mismo serán las que se obtengan de ubicar el siguiente valor del camino en cada una de las casillas que se pueda colocar desde la posición actual. Se parte de la posición donde está ubicado el valor mínimo del tablero y los casos base son aquellos en los que el próximo valor del camino es el máximo número y se está en una posición vecina a su ubicación.

Claro que se puede modificar el procedimiento para que la recursión termine cuando se encuentre una solución, pero se procedió de esta manera para evaluar la

```

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. Segmentos de las funciones mediante las que se determinan los valores prefijados.

estrategia de prefijar los valores del Hidato. De cualquier manera, el problema de resolver un Hidato se puede reducir al Problema de Satisfacción Booleana (SAT, conocido así en inglés por su abreviatura) el cual es considerado un problema NP-completo[2].

5. Inicio del “juego”

Mediante la función `hidatos_list` con signatura `hidatos_list :: Int ->Int ->Int ->String` (`hidatos_list n len ri`) se especifica que se desea generar y solucionar n Hidatos de tamaño len . Mediante funciones definidas en `Utils.hs` se imprimen el Hidato y todas sus soluciones. Al valor ri en este llamado debe asignársele valor 0 como se observa en la Figura 9 para comenzar a utilizar la secuencia de números aleatorios desde el comienzo.

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)

```

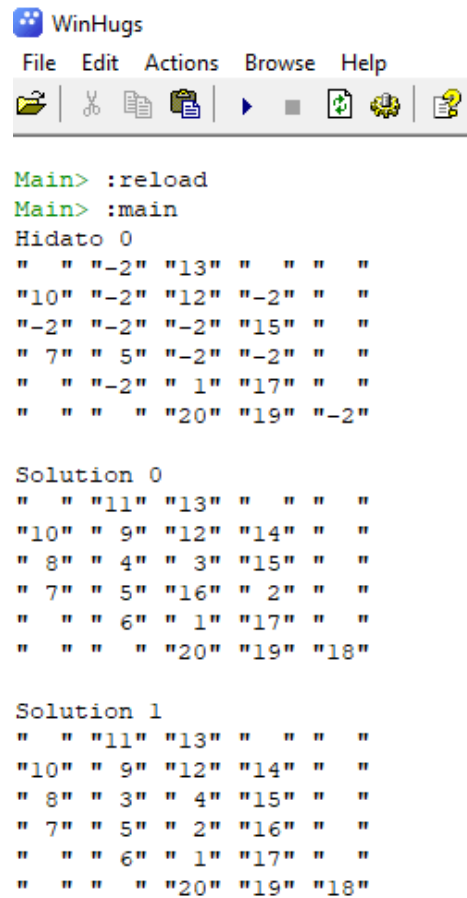
Figura 8. Segmentos de las funciones mediante las que se les da solución al Hidato.

```

main =
    do
        putStr (hidatos_list 5 20 0)

```

Figura 9. Ejemplo de invocación al “juego” en Main.hs.



```
WinHugs
File Edit Actions Browse Help
[Icons]

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. Sección de la salida del “juego” con la invocación de la Figura 9.