# Proyecto de Programación Declarativa

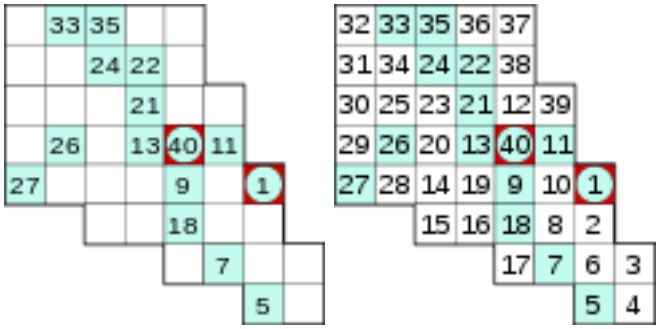
Adrian González Sánchez C412
https://github.com/adriangs1996

Eliane Puerta Cabrera C412
https://github.com/NaniPuerta

November 4, 2020

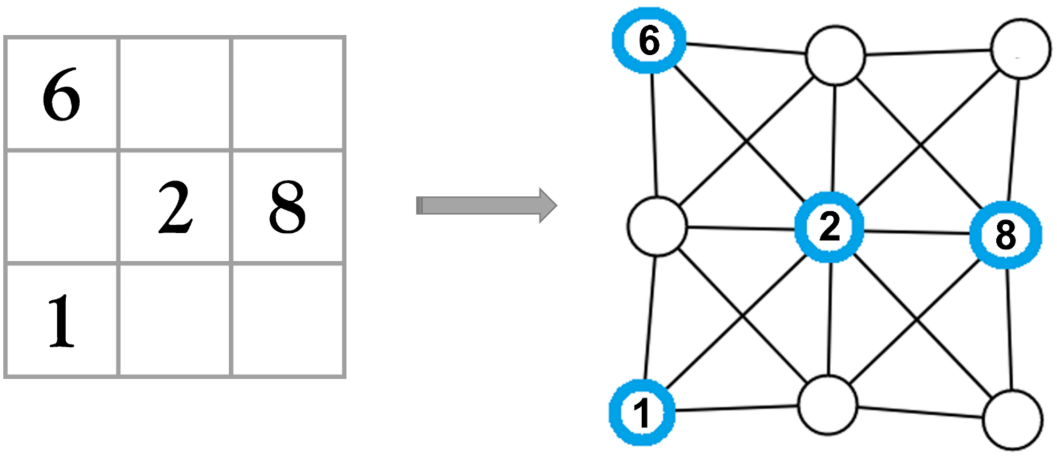
## Orientación

En el Sudoku Hidato, el objetivo es rellenar el tablero con números consecutivos que se conectan horizontal, vertical o diagonalmente. En cada juego de Hidato, los números mayor y menor están marcados en el tablero. Todos los números consecutivos están adyacentes de forma vertical, horizontal o diagonal. Hay algunos números más en el tablero para ayudar a dirigir al jugador sobre cómo empezar a resolverlo y para asegurarse de que ese Hidato tiene solución única. Se suele jugar en una cuadrícula como Sudoku pero también existen tableros hexagonales u otros más irregulares con figuras como corazones, calaveras, etc. Cada puzzle de Hidato creado correctamente debe tener solución única.




## Modelación

Antes de atacar el problema de generar los tableros y resolverlos, es necesario tener un modelo computacional del juego. Al ser un juego de tablero, y las restricciones que imponen el juego, lo más adecuado consideramos que sea un grafo donde los vértices son las casillas del tablero. Cada vértice representado por una coordenada $(x, y)$ está conectado con su 8-vecindad, a excepción, por supuesto, de las casillas de las esquinas y bordes. Además a cada casilla del tablero debemos asignarle un label con un número $x \in [1, 2, ..., n]$ donde $n$ es el número de casillas disponibles en el tablero. Estos labels vienen prefijados en las casillas que conocemos su valor. Dicho esto, el problema de resolver el Sudoku Hidato se traduce en encontrar un camino Hamiltoniano en este grafo y asignar a cada vértice sin label en este camino, un número de forma que aparezcan de manera ordenada y creciente todos los valores $[1, 2, ..., n]$.




Desde el punto de vista programático, definimos dos estructuras, una a utilizar en la solución del problema, y la otra a utilizar en la generación. El punto de unión de ambos algoritmos es el método **boardToString** el cual es utilizado por el generador para devolver un **[String]** que luego puede ser consumida por el método **makeBoard** que devuelve un **BoardProblem** que es la estructura usada para resolver el juego.

## Resolución de un tablero

En el archivo **src/Lib.hs** encontramos todas las definiciones necesarias para crear el método de resolución. Se define la estructura **BoardProblem** que engloba los datos necesarios sobre el tablero:

1. **cells**: Representa las celdas del tablero. Es un diccionario anidado de enteros de modo que podemos acceder a la casilla (x, y) de la siguiente forma: $dict[x][y]$.
2. **onePos**: Posicion del 1 en el tablero.
3. **endVal**: Máximo valor del tablero.
4. **givens**: Valores prefijados del tablero. Estos valores son inicialmente insertados en **cells**.

En este módulo se presentan además varias funciones auxiliares que permiten insertar un valor en el tablero (**tupIns**), obtener el valor de la casilla (x,y), imprimir en consola un tablero y parsear una lista de strings y convertirlas en un tablero. Una función fundamental es **isSolved** la cual determina si un tablero ya está resuelto comprobando si no existen celdas vacías y que se pueda llegar desde el 1 hasta **endVal** en un camino Hamiltoniano, incrementando el valor buscado en cada momento.

El corazón de la resolución es la función **bruteForceHidato** que se encuentra en el módulo **src/Hidato.hs**. Esta función tiene la siguiente signatura:

```
bruteForceHidato ::  BoardProblem -> [IntMap (IntMap Int)]
```

la cual devuelve una lista de tableros pues el algoritmo no requiere que los tableros tengan solución única, es perfectamente capaz de encontrar todas las soluciones del tablero que se le pase. Por supuesto, a la hora de engranar la aplicación, garantizamos la generación de tableros únicos y un paso importante en este proceso es pedir soluciones a demanda, aprovechando el sistema lazy de evaluación de Haskell, de este algoritmo sobre un tablero específico y si en algun momento obtenemos más de una, sabemos que ese tablero no tiene solución única y por tanto no es válido.

La implementación de esta función es básicamente un DFS empezando por las coordenadas del 1 (guardadas en **onePos**), o sea intentamos en cada paso, que nos encontramos en un vértice con label $l_i$, asignarle a algún vecino de dicho vértice que no tenga label, el label $l_{i+1}$ y luego pasar a visitar ese vecino. Hay varios casos:

1. El tablero está resuelto, en dicho caso se devuelve.
2. El vértice que estamos visitando no tiene vecinos disponibles, en dicho caso se devuelve una lista vacía (no es solución).
3. El label que queremos colocar se encuentra en alguno de los vecinos del vértice que estamos visitando, en dicho caso pasamos a visitar ese vecino e intentamos colocar el label $l_{i+1}$.
4. Recursivamente insertamos el label que queremos colocar en algún vecino disponible y pasamos a visitar cada uno de esos vecinos.

El algoritmo solo explora las formas válidas de colocar los labels, además intenta construir un camino válido lo más largo posible, lo que en tableros con solución única, lleva a encontrar la respuesta de una manera bastante rápida, permitiendo resolver tableros decentemente grandes.

Los siguientes ejemplos fueron generados y resueltos en aproximadamente 0.00104s en una Dell Inspiron-7548 con 12GB de RAM y un Intel COREi7 de núcleo. Corresponden a tableros de 100 y 81 casillas respectivamente.

```
    $ ./SudokuHidato-exe -s
            30  0
         35 33  0  0
      38  0 34  0 27  0
   40  0  0        25  0 22
43  0  0              23  0 20
 0 45  0              17 18  0
   47 48  0        13 15  0
       3  0  1 11  0  0
          0  7  9  0
             0  0

            30 29
         35 33 31 28
      38 36 34 32 27 26
   40 39 37        25 24 22
43 42 41              23 21 20
44 45 46              17 18 19
   47 48  2        13 15 16
       3  4  1 11 12 14
          5  7  9 10
             6  8

    $./SudokuHidato-exe -s
       0  0 29  0 26
      33  0  0  0  0
      36 35  0
      38  0  0        0  2
44 43 41  0  0  0  0  0  1
 0  0  0  0  0 18 10  0  0
 0 48  0  0  0  0  0  0  0
```

```
50  0                    0  0
52 53                   13  0

       32 31 29 28 26
       33 34 30 25 27
       36 35 24
       38 37 23         3  2
44 43 41 39 22  9  8    4  1
45 46 42 40 21 18 10    7  5
49 48 47 20 19 17 16 11  6
50 51                   15 12
52 53                   13 14
```

# Generación

La generación de tableros es quizás la parte más interesante del proyecto. Partamos primeramente de la base de que necesitamos que cada tablero tenga una solución única (de lo contrario, generar una matriz y llenarla arbitrariamente garantizando que siempre exista solución es un trabajo trivial). La idea general para generar los tableros se basa en:

1. *Generar una matriz de $NxM$ casillas.*
2. *"Tachar" algunas casillas para darle alguna forma al tablero.*
3. *Ubicar aleatoriamente el 1 y el máximo valor del tablero.*
4. *Hallar un camino de Hamilton con las labels correctamentes colocados.*
5. *Eliminar algunos labels del tablero.*
6. *Comprobar que el tablero tiene solución única.*

La implementación de estos *guidelines* se consolida en el módulo **src/HidatoBoard.hs**, específicamente con la función:

```
generateBoard :: Int -> [String]
```

cuyo párametro es un entero que utilizamos como semilla para generar los números aleatorios y su resultado es una lista de strings con el siguiente formato:

1. Cada fila del tablero corresponde a un string de la lista.
2. Cada casilla de la fila se separa en el string por un espacio.
3. Los posibles elementos de cada casilla son :
   (a) ".": Indica que la casilla es inválida (está tachada, estas se muestran en consola como espacios en blanco).
   (b) "0": Indica que esta casilla está vacía (Hay que ponerle su label).
   (c) X: Donde X es cualquier entero.

**generateBoard** primeramenta selecciona un tablero aleatoriamente de una lista de *templates* definidas previamente por nosotros:

```
generateBoard randSeed = boardToStrings (makeHidatoBoard brd randSeed)
 where
   brd = selectBoard (genRandInt randSeed)
```

aunque pudiéramos realizar esta fase de una forma completamente aleatoria (este proceso corresponde a los tres primeros pasos definidos en los *guidelines*), seleccionar de un conjunto de templates nos permite crear tableros con formas bien definidas (Cruces y rombos, por ejemplo), lo que hace un poco más atractivos los tableros generados. Estos templates no son más que las dimensiones del tablero ($N$ y $M$) con la cual se genera una matriz completamente vacía (todos los valores en 0), y una lista de pares $(x, y)$ representando un conjunto de posiciones tal que, el tablero en la posicion $(x, y)$ está tachado.

Con el tablero vacío en mano, pasamos al próximo paso del algoritmo: Generar un camino Hamiltoniano con los labels correctamente colocados, de lo cual se ocupa **makeHidatoBoard**.

Espera un segundo..., ese problema es el mismo que el de resolver el Hidato !?. La verdad es que no, en este caso, solo necesitamos encontrar *un* camino Hamiltoniano a partir de una posición original, luego los labels los podemos colocar en orden ascendente y tenemos nuestra respuesta.

Espera de nuevo..., encontrar un camino Hamiltoniano es NP-Hard verdad !?. True, pero en este caso, este grafo es basado en una matriz, conecatdo en 8-vecindades (como un tablero de Ajedrez ;) ), y el problema de recorrer este tablero en un camino Hamiltoniano es conocido como **Knight's Tour**, el cual se soluciona en tiempo $O(n)$ con la heurística de **Warnsdorff**. Esta heurística es un caso típico de *greedy* en el cual el tablero se divide en cuadrículas más chicas de acuerdo a la cantidad de movimientos disponibles del caballo (en el caso del Knight's Tour Problem) y en cada momento, se mueve hacia la casilla que permite la menor cantidad de movimientos posibles y que no haya sido visitada. Para poder aplicarlo en nuestro problema, debemos generalizar esta heurística, y una generalización inmediata es siempre moverse al vértice de menor *degree*. Esta es básicamente la idea detrás de la función:

```
buildTreeWarnsdorff
 :: PathTree Int
 -> Board
 -> Coordinates
 -> Int
 -> Int
 -> Int
 -> (Int, PathTree Int)
```

la cual toma los módulos **src/BoatdGen.hs, src/PathTree.hs, src/MyMatrix.hs, src/Solve.hs** para poder calcularse (junto con otras funcionalidades inherentes a las estructuras definidas para hacer más cómoda la implementación).

Una vez que rellenamos el tablero con el camino Hamiltoniano, pasamos a los últimos dos pasos de los *guidelines*, o sea, vaciar algunas casillas, y garantizar que la solución sea única. Para lograr esto, la función

```
getUniqueBoard :: (Eq a) => Board -> PathTree a -> Int -> Board
```

selecciona aleatoriamente una coordenada de la lista de elementos incorporados al tablero (en esta lista, no aparecen las coordenadas del 1 ni del máximo valor del tablero), la elimina y luego corre el método **bruteForceHidato**, del cual intenta extraer 2 soluciones, aprovechándose del sistema

lazy de evaluación de Haskell, si la cantidad de soluciones es 1, entonces es seguro vaciar esa casilla, de lo contrario, el valor de esa casilla se restaura, y esa coordenada se elimina de la lista de coordenadas. Este proceso se ejecuta mientras la lista de coordenadas no quede vacía.

Una vez que tenemos nuestro tablero generado, se le pasa a la función

```
boardToStrings :: Board -> [String]
```

que devuelve la lista de strings necesarias para luego poderse parsear, y ser utilizada por el solver.

# Implementación de la app

El módulo **app/Main.hs** se ocupa de engranar los algorimos implementados en una simple interfaz de usuario basada en líneas de comando para correr tanto el generador, como el solver. Para implementar esta funcionalidad, hicimos uso extensivo del **IO Monad** junto con el módulo **GetOpt**

```
import          System.Console.GetOpt
```

donde usamos principalmente la función **getOpt** para parsear los argumentos devueltos por **getArgs** y agruparlos en flags, de acuerdo las opciones definidas en *options*, de la siguiente forma:

```
   parse :: [String] -> IO [Flag]
parse argv = case getOpt Permute options argv of
    (args, _, []) -> do
        if Help `elem` args
            then do
                hPutStrLn stderr (usageInfo header options)
                exitWith ExitSuccess
            else return (nub args)

    (_, _, err) -> do
        hPutStrLn stderr (concat err ++ usageInfo header options)
        exitWith $ ExitFailure 1
    where header = "Usage: SudokuHidato [-rsftg]"
```

# Compilar el proyecto y usar la app

El proyecto usa el sistema de compilado para Haskell **stack**. Obtener un compilado debe ser tan fácil como ejecutar los siguientes comandos:

```
$ stack setup
$ stack build
```

Además proveemos un compilado SudokuHidato-exe (**Linux**), que se puede utilizar en caso de que no se quiera compilar el proyecto. Esta app es un programa de línea de comandos que ofrece las siguientes opciones:

```
$ ./SudokuHidato-exe --help
Usage: SudokuHidato [-rsftg]
          --help                Print this help message
-g COUNT  --generate=COUNT      Generate COUNT boards and shows them
```

```
  -s        --genandsolve         Generate 1 board, prints it, solves it and
                                  prints the solution
  -t FILE   --gentestfile=FILE    Generate 10 boards and save them in FILE
  -f FILE   --solvefromfile=FILE  Read boards in file and solve them
  -r FILE   --showFile=FILE       Read boards in file and show them
```

Con el flag **-g** podemos generar una cantidad arbitraria de tableros y los imprime en consola para resolverlos a mano o algo por el estilo. La generacion de tableros ocurre cada 1 segundo, pues el tiempo actual se usa como semilla para la generación de numeros aleatorios. Por ejemplo:

```
    $ ./SudokuHidato-exe -g 3
       0 10  8  0  0
       0 12  0  0  5
      15 14  0
       0 17  0        0 43
 0  0  0  0  0  1  0 45  0
23  0 32  0  0 38  0  0 47
 0 30  0 34 36  0  0 48  0
26  0                 53  0
 0 28                  0 52


            47 48
          0  0  0 44
       0 39  0 42  0  0
    0 36 38        0  5  1
 0  0 34              0  0  0
 0 31 32             11  0  3
    0  0  0        0 12  0
      26  0 22  0 15 13
         23  0 18 16
             0 19


       0 24 26  0  0
       0  0  0  0 30
       0  0  0
       0 34  0        0  0
 7  0  0  0 36  0  0 43 42
 0  9  0 15  0 38 46  0  0
 0  0  0 12  0  0  0 48  0
 0  4                 0 52
 1  2                53  0
```

Con el flag **-s** se genera un tablero, se muestra y se soluciona. Por ejemplo

```
    $./SudokuHidato-exe -s

        0
```

```
 0  0  0
0 18 15 12  0
 0  0  0  1  0  0
       0  0  2

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

Las otras tres opciones sirven para generar tableros y guardarlos para luego utilizarlos como entrada para el algoritmo solucionador. Por ejemplo, con el flag **-t** generamos 10 tableros y los guardamos en un archivo que se pasa como entrada en la línea de comandos. Este archivo luego puede ser mostrado o resuelto usando los flags **-f** o **-r**: Por ejemplo:

```
$ ./SudokuHidato-exe -t test.txt
$ ./SudokuHidato-exe -r test.txt
             0 19
          23 21  0 16
       0 24  0  0 15  0
   28 27 25       14 12  0
31  0  0             10  9  0
 0 33  0              0  0  5
   35  0  0        1  3  0
      37  0 41  0  0  0
         40 48 47  0
            0  0

       0
 4  0  1
 6  0

       0
    0 10 12
 6  0 11  0 14
    0  0  3  0 17 18
          0  1  0

    ......... omitidos los otros 7 por brevedad .........
```

```
$ ./SudokuHidato-exe -f test.txt
            20 19
          23 21 18 16
       26 24 22 17 15 13
   28 27 25       14 12 11
31 30 29             10  9  6
32 33 34              8  7  5
   35 36 38        1  3  4
      37 39 41 42 43  2
         40 48 47 44
            46 45
```

```
      2
4  3  1
6  5

      9
   8 10 12
6  7 11 13 14
   5  4  3 15 17 18
         2  1 16
.............. omitidos los otros 7 por brevedad ............
```

Por supuesto, si se quieren utilizar más casos de prueba en un archivo, se puede hacer uso de los poderes de **bash**.