**Proyecto de Haskell**
**Curso: 2019-2020**
**Título: Sudoku Hidato**

Oscar Luis Hernández Solano o.hernandez2@estudiantes.matcom.uh.cu
Grupo C411

Harold Rosales Hernández h.rosales@estudiantes.matcom.uh.cu
Grupo C411

# 1 Juego:

Hidato es un juego de lógica creado por el Dr. Gyora Benedek, matemático israelí. El objetivo de Hidato 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.

# 2 Aspectos generales:

El proyecto consta de dos partes fundamentales, la primera consiste en encontrar la solución para un tablero de Hidato de cualquier forma y bien definido según las reglas descritas en la sección anterior y la segunda es crear un generador de tableros de Hidato con solución. Para ello modelamos el tablero con una estructura de matriz que no es más que una lista de listas de enteros, en la que en cada posición va a haber un número entero $n \geq 1$ si está ocupada, $n = 0$ en caso contrario y $n = -1$ en caso de que la casilla no forme parte del tablero de juego.

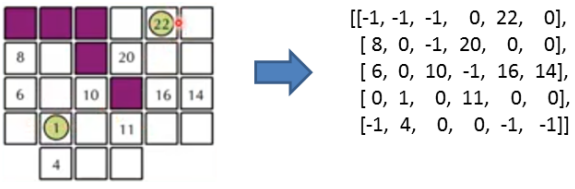


Figura 1. Representación de un tablero de Hidato.

# 3 Solución:

La estrategia seguida para encontrar la solución de un Hidato es la de buscar todas las soluciones y filtrar las soluciones correctas, para ello implementamos un algoritmo tipo Backtraking descrito en pseudocódigo a continuación. Sean:

$n$ : El número actual.
$brd$ : Un tablero de Hidato.
$(i, j)$ : Una posición en el tablero.
*givens*: La lista de números dados inicialmente.

```
solve(brd, (i,j), n, givens):
    CASE 1: if n > last(givens) then brd
    CASE 2: if n == first(givens)
                and not n in aroundsOf(brd,(i,j)) then []
            else solve(brd, (ni, nj), n+1, tail(givens))
    CASE 3: for (row,col) in aroundsOf(brd, (i,j)):
            if brd[row, col] == 0 then
                solve(brd, (row, col), n+1, givens)
```

El caso número 1 es de parada estamos intentando colocar un número mayor que el mayor del tablero lo que significa que se consiguió colocar el resto de números anteriores correctamente, en dicho caso retornamos la solución encontrada. El caso número 2 verifica si el número que se quiere colocar es uno de los que estaban prefijados inicialmente, en caso de que sea esto positivo para que siga siendo una solución válida el número $n$ tiene que estar en los adyacentes dígase horizontal, vertical o diagonalmente de el elemento en la posición $(i, j)$ porque es su sucesor, en caso de que no se encuentre a $n$ en los adyacentes se poda el árbol regresando en la recursividad y en caso de que sí se encontrara entonces se llama recursivamente en la posición de $n$ a colocar $n+1$ y se avanza en la lista de los valores dados; aclarar aquí que la lista de los números dados está ordenada de menor a mayor, mismo orden en que se van colocando los números en el tablero. En el caso 3 se esta intentando colocar un número $n$ que no está entre los fijados al inicio y es menor que el mayor del tablero, luego se buscan todas las posiciones vacías adyacentes a $(i, j)$ en el tablero, se coloca a $n$ en dicha posición y se intenta resolver el tablero resultante llamando recursivamente a colocar $n + 1$ en la nueva posición, con el nuevo tablero, la lista de dados tal cual. El algoritmo descrito es correcto porque se parte del valor inicial y en todo momento se va moviendo por celdas adyacentes colocando y/o encontrando al sucesor de dicho número hasta llegar al mayor valor del tablero, si en algún momento no es posible colocar o encontrar adyacentemente al sucesor del valor actual entonces se retorna una lista vacía y dicho algoritmo termina porque en cada iteración se incrementa en uno el valor de $n$ asegurando que en algún momento vaya a ser mayor que el mayor del tablero que es un número entero positivo y según las reglas siempre fijado al inicio del juego.

# 4 Generación:

Sean:

*brd*: $[((i_0, j_0), val_0), .., ((i_n, j_n), val_n)]$

*shapes*: Lista de tableros de distintas formas y dimensiones con todas las celdas en cero.

PASO 1: Generar un número random $0 \leq r < length(shapes)$ para escoger la forma del tablero.

PASO 2: Generar dos números random $0 \leq n_1, n_2 < length(brd); n_1 \neq n_2$ para las posiciones de mayor y menor número en el tablero.

PASO 3: Solucionar el tablero con solo el primero y último valor fijados utilizando la función descrita en la sección anterior.

PASO 4: Generar una lista de números random $[k_1, k_2, ..., k_{\frac{2*length(brd)}{3}}]$ con $0 \leq k_i < length(brd)$ para ser las posiciones a convertir en 0 en el tablero resuelto, notar en este paso que nunca se escogerán el menor ni el mayor número del tablero para hacerse cero y no violar así las reglas del juego.

Para la generación de tableros de Hidato con solución implementamos el algoritmo descrito por los pasos anteriores nótese que la forma de los tableros no se genera, se escoge aleatoriamente de una lista de tableros con distintos tamaños y formas y se escogen para hacerse ceros dos tercios del total de celdas del tablero, sería interesante poder variar el número de celdas a convertir en cero creando una suerte de nivel de dificultad del Hidato en cuestión para el jugador. La cantidad de celdas a hacerse cero no siempre es exactamente $\frac{2*length(brd)}{3}$ ya que por la manera en que generamos números aleatorios pueden quedar índices con el mismo valor lo que significa escoger la misma celda para hacerse cero. A continuación les presentamos un ejemplo de cómo creamos números aleatorios.

```
import Data.Maybe
import System.Random

randomlist :: Int -> StdGen -> [Int]
randomlist n = take n . unfoldr (Just . random)

-- lista de k enteros random congruentes modulo n
foo n k = map (\x -> x `mod` n ) ( randomlist k seed )
```

Vale destacar que los tableros de Hidato que estamos generando tienen solución pero no garantizamos su unicidad. Si se modifica la función que encuentra la solución de un tablero a que en lugar de que en cuanto encuentre una solución retorne, calcule una lista de todas las soluciones posibles, entonces y una propuesta para generar tableros que tengan solución única sería la siguiente:

PASO 1: Generar un tablero de Hidato.

PASO 2: Calcular todas las soluciones.

PASO 3: Si tiene una sola solución $\Longrightarrow$ FIN. En caso contrario encontar una posición en la que la mayoría de los tableros difieran y fijar esa posición $\Longrightarrow$ PASO 2.