Proyecto de Programación Declarativa

Mauricio L. Perdomo Cortés Marcos Antonio Maceo Reyes

October 31, 2020

Facultad de Matemática y Computación
Universidad de La Habana

1 Orientación

Sudoku Hidato:

El proyecto tiene dos objetivos generales:

- Crear un programa en Haskell que dado un Hidato con algunas casillas anotadas sea capaz de darle solución.
- Crear un programa en Haskell que sea capaz de generar Hidatos.

2 Ideas generales

A continuación están explicadas diferentes ideas y conceptos que están presentes en la solución del ejercicio:

- **Board:**
  Un tablero o **Board** es el tipo que representa los distintos Hidatos en nuestro programa. Este tipo está conformado por el **data constructor Board** que recibe dos argumentos uno de tipo **Data.Vector BoardTile** y otro de tipo **Int**. Los tableros estan compuestos por un **Vector BoardTile** (el tipo **BoardTile** será explicado más adelante) que representa las celdas del tablero y un **Int** que representa el largo de una fila. Los tableros son rectangulares por lo tanto la cantidad de elementos en el vector siempre es un múltiplo de la longitud de la fila.
- **BoardTile:**
  BoardTile es el tipo que representa cada celda del tablero o **Board**. Este tipo es la "suma" de dos **data constructors Empty** y **Value**, el primero recibe un **Int** que representa el índice del **Vector** del tablero donde se encuentra esta celda, los valores creados con este constructor representan las celdas del tablero que no tienen que ver como tal con el Hidato. El segundo (**Value**) recibe dos argumentos de tipo **Int** el primero tiene la misma función que el primero de **Empty**, el segundo representa el valor de esta celda, en el caso de que este valor sea 0 significa que la celda no tiene número asignado, este constructor se usa para crear las celdas que sí forman parte del puzzle.

- **Template:**
  Los **templates** son usados para la generación de tableros, estos constituyen la forma de un tablero, el proceso de generar un tablero consiste en tomar un template y asignarle números a algunas de sus casillas de modo tal que exista una única forma de asignar números a las celdas vacías (celdas representadas por valores de la forma **Value _ 0**). Más adelante explicamos algunas restricciones establecidas sobre los **templates** para la generación de tableros.

- **Puzzle:**
  Cuando mencionamos **puzzle** nos referimos al conjunto de celdas del tablero que tendrán números cuando el juego haya finalizado, son aquellas celdas que conforman el **Hidato** como tal.

3 Módulos.

En esta sección abordaremos los módulos que conforman nuestra librería **HidatoLib**

3.1 Board

En este módulo se encuentran definidos los tipos que representan nuestro tablero, los cuales fueron mencionados anteriormente, a continuación procederemos a dar una breve explicación de algunas de las funciones que conforman el módulo

computeNeighbors :: Board -> BoardTile -> [BoardTile]
Esta función recibe un tablero y un celda y nos devuelve una lista con todas las celdas vecinas a esta, incluye las celdas que forman parte del **puzzle** *y las que no.*

updateBoardValues :: Board -> [BoardTiles] -> Board
Esta función recibe un tablero y una lista de celdas y devuelve un nuevo tablero modificando el anterior con las celdas de la lista.

randomizePathNTimes :: Board -> Int -> [BoardTile] -> IO [BoardTile]

Esta función juega un papel muy importante en la generación de tableros, lo que hace es dado un tablero y un camino de celdas en ese tablero, transforma este camino usando una estrategia que llamamos **backbite** *que explicaremos más adelante. La función llama N veces a la función* **randmomizePath**.

computePathForBoard :: Board -> [BoardTile]
Otra función con un papel primordial en la generación de tableros, hablaremos de esta con más profundidad en otra sección del informe. La función recibe un tablero y devuelve un camino que abarca todas las celdas de nuestro **puzzle**, *el tablero debe cumplir las restricciones impuestas a los* **templates**. *La función depende enormemente de la función* **nextIndex** *de la cual hablaremos más en otra sección.*

searchValue :: Board -> Int -> Maybe BoardTile
Esta función recibe un tablero y un valor y devuelve la celda del tablero donde está este valor, en caso de que el valor no esté en el tablero devuelve **Nothing**.

rotateBoard :: Board -> Board
Esta función recibe un tablero y devuelve el tablero rotado 90 grados en sentido contrario de las manecillas del reloj.

3.2 Graph

En este módulo se encuentran funciones y tipos para interactuar con grafos. Este módulo tiene gran importancia en nuestro solucionador de **Hidatos**.

type Graph
Este es el tipo que usaremos para representar los grafos, es solamente un nuevo nombre para **Map Int (Set BoardTile)**, *que vendría siendo las listas de adyacencia del grafo.*

fromBoardToGraph :: Board -> Graph
Esta función crea un grafo teniendo en cuenta solo las celdas del **puzzle**. *Dos nodos serán adyacentes en el grafo si cumplen que son vecinos en el tablero (esto se obtiene con la función* **computeNeighbors** *mencionada anteriormente) y son números consecutivos ó si son vecinos en el tablero y*

uno es 0 y el otro es un nodo que no tiene como vecino a su antecesor y sucesor.

searchPathOfN :: Board -> Int -> IntSet -> (Int, Int) -> Maybe [[Int]])
Es la función principal para dar solución al **Hidato** *también tiene importancia en la generación de nuevos* **Hidatos**, *esta función dado un tablero, un N y dos celdas del tablero nos devuelve todos los caminos simples de longitud N entre ese par de nodos. Abordaremos con mayor profundidad está función en otra sección.*

3.3 Parse

En este módulo se encuentran las funciones para desde un fichero obtner el tablero o **Board** asociado. Existen dos funciones porque el formato que tiene un template en un fichero es ligeramente diferente al formato que tiene un **Hidato**.

parseBoard :: String -> IO Board
Esta función recibe el nombre de un fichero donde se encuentra un **template** *y devuelve un* **Board** *que lo representa. Para parsear el contenido del fichero dividimos este por líneas y parseamos cada una luego concatenaremos los vectores resultantes para crear el vector que contiene todas las celdas del tablero. Parsear un línea es muy sencillo ya que cada caracter representa una celda y puede ser '-' ó '0', en el primer caso significa que la celda no pertence al* **Hidato**, *en el segundo caso sí pertenece.*

parseGame :: String :: IO Board
Esta función recibe el nombre de un fichero donde se encuentra un **Hidato** *y devuelve el* **Board** *que lo representa, funciona de forma muy parecida a* **parseBoard** *con la diferencia de que en este caso las celdas de una fila estarán separadas por espacios y pueden estar conformadas por más de un caracter.*

3.4 Print

En este módulo se encuentran las funciones para obtener el **String** que representa un **Board**.

printBoard :: Board -> String
Esta función devuelve el **String** *que representa el tablero que recibe como argumento, para esto divide el vector que representa al tablero en vectores con el tamaño de las filas del tablero (recordemos que este valor se pasa como argumento al* **data constructor Board***) y obtiene el* **String** *que representa a cada una de estas filas.*

3.5 Utils

En este módulo se encuentran varias funciones que cumplen propósitos más generales por lo que no fueron puestas en ninguno de los módulos anteriores.

mergeSort :: (a -> a -> Bool) -> [a] -> [a]
Está función es una implementación del conocido algoritmo merge sort que recibe la una función que utilizará para comparar los elementos.

groupInN :: Int -> [a] -> [[a]]
Esta función recibe un **Int** *N y una lista y divide la lista en varias listas de tamaño N, es posible que la última lista no llegue a N elementos.*

groupConsecutives :: [Int] -> [[Int]]
Esta función recibe una lista y agrupa los valores que sean consecutivos y sean adyancentes en la lista, por ejemplo si le pasamos como argumento **[1,2,3,5,6,9,11,12]** *obtendremos* **[[1,2,3],[5,6],[9],[11,12]]**.

findGaps :: [Int] -> [[Int]]
Dada una lista de **Int** *encuentra los pares de números consecutivos que no tienen sus valores consecutivos, por ejemplo si le pasamos como argumento* **[1,2,4,6,7,8,9]** *obtendremos* **[[2,4], [4,6]]**.

myMaybeLiftA2 :: (a -> a -> a) -> Maybe a -> Maybe a ->

Maybe a
Esta función es muy similar a liftA2 que se encuentra en Control.Applicative, la razoón por la que existe es que el la implementación de **Applicative** *del tipo* **Maybe** *no funcionaba para nuestro objetivo.*

divideVector :: Vector a -> Int -> Vector a
Esta función realiza la misma acción que **groupInN** *pero aplicada a vectores.*

4 Generar Hidatos

En esta sección explicaremos como generamos **Hidatos**. El proceso que utilizamos para crear un **Hidato** puede resumirse en lo siguiente: tomamos un **template** que cumpla determinadas condiciones, creamos un camino sobre este **template**, luego modificamos un poco este camino para que sobre el mismo **template** no se genere siempre el mismo **puzzle** y luego retiramos algunos de los números. En esta sección abordaremos con más profundidad como ocurren cada uno de los pasos anteriores y al final haremos un pequeño análisis sobre los pensamientos e ideas que nos llevaron a tomar estas decisiones.

4.1 Templates

Como observamos en los pasos explicados la generación de un tablero se basa en un **template**, un **template** es un fichero de texto que contiene la representación de un tablero, estará representado de forma rectangular y las celdas que formen parte del **puzzle** seran '0' y las que no '-'. Para que el comportamiento del generador sea el esperado el **template** debe cumplir que pueda crearse un camino que abarque todas las celdas del **Hidato** usando la estrategia explicada a continuación.

4.2 Crear el camino

El proceso de creación del camino es el siguiente, se busca la celda que pertenezca al **puzzle** que se encuentre en la menor fila que contenga celdas del **puzzle**, si hay más de una en esta fila se toma la que esté más a la izquierda, desde esta celda se avanza hacia la derecha hasta que no exista

en la derecha una celda que pertenezca al **Hidato**, en este momento se desplazará hacia la fila inferior a la celda más a la derecha que sea vecina de la actual y se repetirá el proceso pero esta vez avanzando hacia la izquierda.

4.3 Modificar el camino

Si sobre un **template** siempre generamos el mismo camino, tendríamos solo un **Hidato** por **template** para solucionar esto decidimos modificar el camino creado, para modificarlo usamos un movimiento conocido como **backbite**(para una mayor información sobre esto puede ir a `https://arxiv.org/pdf/cond-mat/0508094.pdf`) el movimiento consiste en seleccionar uno de los dos extremos del camino y luego seleccionar uno de los nodos vecinos a este, deben ser vecinos adyacentes a él, colocamos una arista entre ambos nodos y luego retiramos del camino la arista mediante la cual el nodo vecino llegaba al nodo extremo elegido, por ejemplo supongamos que tenemos el camino [0, 1, ..., i, i+1, i+2, ..., i+n] digamos que el vértice i y el vértice i+n son vecinos nuestro nuevo camino quedaríamos [0, 1, ..., i, i+n, i+(n-1), i+(n-2), ..., i+1]. Este procedimiento lo realizamos una cantidad aleatoria de veces y terminamos con caminos bastantes aleatorios.

4.4 Eliminar números del tablero

Al llegar a este punto tenemos un tablero con un camino de números generados, para tener nuestro **Hidato** debemos quitar algunos de estos números garantizando que los números que queden guíen únicamente a nuestra solución. Para seleccionar los números que retiraremos primero obtendremos una permutación aleatoria del camino de números e iremos uno por uno quitando el número y resolviendo el tablero resultante, si obtenemos más de una solución esto nos indicará que no podemos eliminar este número en el caso de que obtengamos una única solución quitaremo este números y avanzaremos hacia el próximo.

4.4.1 Análisis

Consideramos que la solución ideal sería poder dar un **Hidato** para cualquier **template** que sea posible, el problema de esto viene dado porque hacer esto es equivalente a encontrar un camino de Hamilton para el grafo

que represente al **template**, esto fue lo que nos impulsó a restringir los **templates** a algunos sobre los que pudiesemos crear un camino "trivial". Para extender un poco la posibilidad de los templates añadimos la opción de rotar un tablero esto nos permite usar **templates** como la flecha que se encuentra en nuestros ejemplos, que para poder ser usada debe construirse de forma horizontal pero luego podemos tener este **puzzle** tanto horizontal como verticalmente. La definición de nuevos caminos "triviales" sería otra forma de ampliar el conjunto de **templates** que se pueden usar.
La complejidad de nuestro algoritmo para generar tableros puede ser analizada por partes, iremos por cada uno de los procesos explicados anteriormente:

- Obtener el **Board** que representa a un **template** es *O(N)* con N la cantidad de celdas del **template**
- Obtener nuestro camino trivial es *O(N)*, modificar un camino es *O(N)* también dado porque se construye un nuevo camino a partir del anterior. En este caso N reprsenta la cantidad de celdas del **puzzle** que son las que pertenecen al camino. Este proceso se realiza entre dos y tres veces la cantidad de celdas del **template** por lo que si M representa este número la complejidad total de "randomizar" nuestro camino sería *O(M*N)*.
- Eliminar números de nuestro **puzzle** es la parte más costosa de nuestro algoritmo, esto viene dado por la necesidad de que el tablero final tenga solución única, como vimos para esto solucionamos el **Hidato** que queda al retirar cada una de las celdas del camino, como veremos a continuación resolver un **Hidato** tiene una complejidad exponencial con respecto al tamaño de la entrada.

5 Resolver Hidatos

Esta es la última sección de nuestro informe, aquí abordaremos el proceso que seguimos para resolver los **Hidatos**.
Como hemos mencionado anteriormente podemos ver que solucionar un **Hidato** es equivalente a encontrar un camino de Hamilton en el grafo que representa al **puzzle** con la diferencia de que sabemos la posición en el camino de algunos de los nodos. Sabemos que no tenemos una solución a

este problema mucho mejor que la fuerza bruta, por lo que nuestra solución será realizar un Backtrack buscando todos los caminos posibles, realizando algunas decisiones que nos permitan reducir los caminos erróneos que tomamos. Para resolver un **Hidato** dado primero buscaremos todos los "gaps" de nuestro **puzzle**, llamamos "gap" a los conjuntos de números consecutivos que deben estar en el tablero pero que actualmente no existen, estos "gaps" son los conjuntos de números que nosotros debemos colocar para resolver el **Hidato**. Ahora podemos plantear nuestra solución como la combinación de soluciones de cada uno de los "gaps" siempre que no exista intersección entre estas dos a dos. Para resolver un "gap" lo que haremos será aplicar la función **searchPathOfN** que nos devolverá una lista con todos los caminos de longitud N entre dos celdas que le digamos y que sean posibles en el **Hidato** actual. Para resolver nuestro **Hidato** ordenaremos los "gaps" por cardinalidad para atacar primeramente a los menores, esto se debe a que el algoritmo para resolver un "gap" es un backtrack con DFS que crece exponencialmente con respecto a la longitud del camino que buscamos y mientras mas "gaps" estén resultos antes menos celdas posibles hay para buscar los caminos, luego por cada una de las soluciones obtenidas para este "gap" intentaremos resolver los "gaps" restantes con el **puzzle** resultante de utilizar esta solución.