Proyecto Evaluativo de Haskell. Curso 2019-2020

Antonio Jesús Otaño Barrera OTANO0398@GMAIL.COM
Grupo C411

Gilberto González Rodríguez GILBECUBANO@GMAIL.COM
Grupo C411

Resumen

Se provee una implementación en Haskell de un programa solucionador de Sudokus Hidato, así como un generador de tableros que devuelve un Hidato con al menos una solución válida.

Abstract

We provide a Haskell implementation of a Hidato puzzle solver and also a Hidato puzzle generator which gives us a puzzle with at least one valid solution.

Palabras Clave: Sudoku Hidato, Haskell
Tema: Programácion funcional, Haskell.

1 Introducción

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.

A partir de estas reglas nos propusimos crear un programa solucionador y otro generador de Sudokus Hidato en el lenguaje Haskell.

2 Ideas generales

Durante el proceso de desarrollo del proyecto pasamos por varias etapas. Se comenzó implementando funciones sencillas para ir aumentando la complejidad a medida que el lenguaje y la técnicas de la programación funcional se hacían más familiares. Una de las primeras características implementadas fue el módulo Printm para imprimir el tablero con un formato amigable. Para ello se utilizó la conocida ténica de programación declarativa de operar con la cabeza de una lista y hacer recursión en el resto de la lista con la lista vacía como caso base. En el proyecto dado un tablero ([[Int]]) cada fila se imprime en una línea y cada valor se imprime separador por espacios y con longitud 3 a pesar de ser un número cuya representación en string tenga longitud 1 o 2.

En el solucionador la primera idea seguida fue la de generar todas las permutaciones del conjunto de valores q faltan en el tablero, insertar la permutación p en las celdas vacías de izquierda a derecha y de arriba hacia abajo y hacer un test para ver si esa permutación era una solución. Por supuesto esto fue impracticable ya para tableros con 5 o 6 celdas vacías.

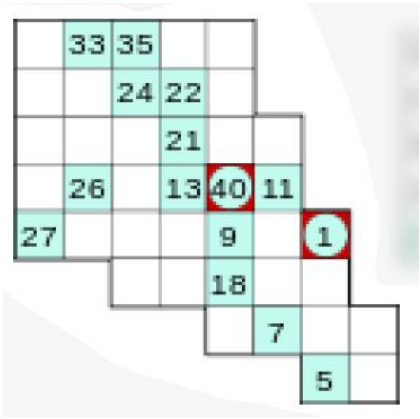
Para el generador la primera implementación realizada consistía en generar de forma aleatoria el principio y el fin a priori y comprobar con el mismo algoritmo solucionador nuestro si había solución, en caso negativo se probaba con otra pareja aleatoria de origen-destino. Esta idea igual resultó impracticable

para tableros de dimensiones no muy grandes.

No obstantes todas estas ideas sirvieron como base y la mayoría del código pudo ser reutilizado en las implementaciones finales.

3 El solucionador

El tablero se puede ver como una matriz, para nosotros en Haskell sería una lista de listas de enteros donde el número 0 indica una posición vacía y un -1 indica una casilla donde no se pueden colocar números (un hueco en el tablero). También se encuentran en el tablero el menor y el mayor número de la solución y algunas casillas con otros números para forzar a construir una solución con determinadas características o para garantizar la unicidad de algunos tableros o para ayudar a un jugador humano, estos números extras también se conocen como pistas. Nuestro programa solucionador debe encargarse de recibir una matriz con estas características, una posición para el inicio, otra para el fin y devolver una posible solución al tablero, si es que este la tiene. A manera de ejemplo, tenemos que para el tablero:



la entrada para nuestro programa sería:

Matriz del tablero:
[[0, 33, 35, 0, 0, -1, -1, -1],
[0, 0, 24, 22, 0, -1, -1, -1],
[0, 0, 0, 21, 0, 0, -1, -1],
[0, 26, 0, 13, 40, 11, -1, -1],
[27, 0, 0 , 0 , 9, 0, 1, -1],
[-1, -1, 0, 0, 18, 0, 0, -1],
[-1, -1, -1, -1, 0, 7, 0, 0],
[-1, -1, -1, -1, -1, -1, 5, 0]]

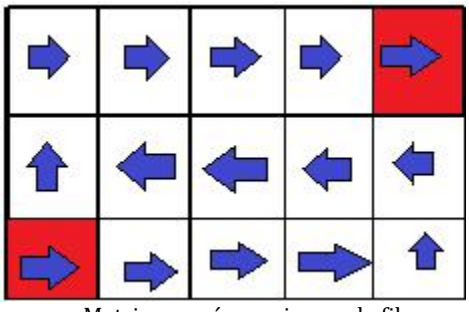
Inicio: (4, 6)
Fin (3, 4)

El algoritmo para solucionar el Hidato se basa en la estrategia de backtracking. Devuelve una tupla donde el primer elemento es un valor booleano que indica si el Hidato tiene solución y en caso positivo el segundo elemento contiene la solución. Comenzamos en la posición inicial y analizamos todas las posiciones adyacentes hacia las cuales nos podemos mover, estas son todas las casillas vacías o las que tienen un valor igual al de la casilla actual más uno. En ese caso vemos si una llamada recursiva al método tomando como posición inicial la casilla adyacente (si era una vacía se llena con el valor que toca) logra resolver el Hidato. Cuando la posición en la que estamos parados es igual a la posición final entonces hemos logrado recorrer el tablero de principio a fin cumpliendo con todas las reglas del juego por lo que nos encontramos con una solución válida y la devolvemos. Nuestro algoritmo se queda con la primera solución válida que encuentre, en caso de no encontrar un adyacente al cual te puedas mover y tras haber agotado todas las posibilidades se devuelve que no hay solución.

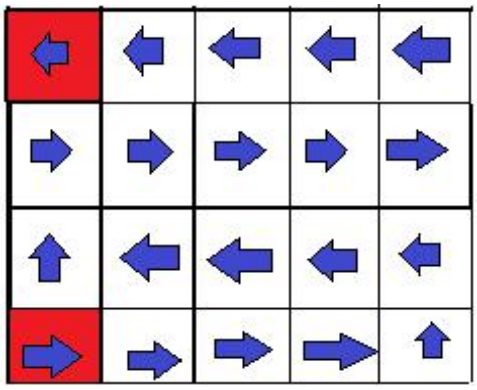
4 El generador

El programa generador se encarga de crear un tablero de Hidato de tal forma que este tenga al menos una solución válida. El número de filas y columnas del tablero se decide de forma aleatoria de tal forma que el tablero más pequeño posible es de 3x3 y el

más grande de 10x10. El generador presenta la limitante de que solo puede generar tableros rectangulares y sin huecos, además por razones de eficiencia no generamos tableros cuya posición de inicio pertenece al conjunto {(1, y) | y ∈ {1..10}} (varios tableros con cierta configuración de filas y columnas como por ejemplo 7x4 demoran mucho con este tipo el inicio en la fila 1, comenzado por 0). Nuestra estrategia se basa en formar una lista inicial con todas las posiciones de la matriz, excepto las antes mencionadas, una de estas posiciones será el principio del Hidato. Por tanto seleccionamos una al azar y mediante backtracking tratamos de construir un camino de números consecutivos que empiece en esa posición y recorra todas las casillas del tablero. Si se logra construir tal camino, entonces el último número de ese camino sería la casilla final de nuestro Hidato. Luego en la matriz donde está escrito el camino sustituimos una cantidad k de valores aleatorios por 0 para crear un sudoku jugable que sabemos que al menos tiene una solución, la que acabamos de construir. El valor de k que escogimos es igual a dos tercios del total de casillas. En el caso de que no se logre construir un camino partiendo de esa casilla, la eliminamos de la lista y repetimos el procedimiento tantas veces como sea necesario hasta encontrar una casilla válida para el inicio. Sabemos que existe al menos una casilla que lo cumple como se muestra a continuación:



Matriz con número impar de filas



Matriz con número par de filas

Por lo cual siempre se devolverá un Hidato con al menos una solución.

4 Complejidad temporal

El problema se puede ver como encontrar un camino de Hamilton en el grafo subyacente, donde cada celda es un nodo y hay una arista entre cada celda adyacente, por lo que este problema es equivalante al clásico problema del viajante. Entonces dado que no se realiza ninguna heurística y en ambos casos (generador y solucionador) se utiliza la estrategia de backtracking sin ninguna poda especial, podemos decir que la complejidad temporal de estos es exponencial con respecto al número de celdas.

5 Cómo ejecutar el programa

Para ejecutar el programa se debe tener instalado el módulo System.Random que puede ser instalado con uno de los siguientes comandos:

```
$ stack install random
$ cabal install random
```

Todos los ficheros de código se encuentran en el directorio **src**. Para ejecutar tanto el generador como el solucionador, ejecutar en Windows los comandos:

```
$ ghc generator.hs
$ .\generator.exe
```

La ejecución del programa creará un sudoku aleatorio y a su vez brindará una respuesta al mismo.

Por otro lado si se quieren realizar varias llamadas consecutivas al programa se puede ejecutar el comando:

$.\run.cmd

También el fichero tester.hs contiene en código la entrada para el caso de prueba de la orientación y computa la solución para este

$ ghc tester.hs
$.\tester.exe

6 Conclusiones

Se implementaron los dos programas, el generador y el solucionador utilizando como estrategia en ambos el backtracking, dándole solución al los requerimientos del proyecto. Con este pudimos notar y aprovechar varias carácterísticas de Haskell y de la programación funcional en general como son la evaluación perezosa y la compresión de listas; así como la definición de funciones por medio del planteamiento de reglas o restricciones (notado principalmente con el uso del where) y la recursión para reducir y aplicar un conjunto de funciones a listas y estructuras.

7 Github link

https://github.com/school-projects-UH/Sudoku-Hidato.git

8 Bibliografía

- Pattern-Based Constraint Satisfaction and Logic Puzzles de Denis Berthier Capítulo 16.2 Path finding Numbix and Hidato
- Real World Haskell de Bryan O'Sullivan
- Conferencias de Haskell
- **Sitios Web**
  - stackoverflow.com
  - haskell.org/documentation