Sudoku Solver como un problema de satisfacción de retricciones con optimización por primer fallo

Luis Alberto Hernández Aguilar

14 de febrero de 2021

Resumen

Para el proyecto final del curso de Programación Declarativa decidí implementar un programa interactivo que permita al usuario crear un tablero para jugar Sudoku, asignando valores a las celdas del tablero, pero que, en vez de que el usuario lo resuelva, lo resuelva el programa y le indique al usuario si la solución del Sudoku es única, en cuyo caso mostraría la solución; o si tiene más soluciones, en cuyo caso mostraría un ejemplo de una. En este documento se habla sobre la estrategia usada para resolver el problema, la implementación y sobre cómo poder ejecutar el proyecto.

1. Preliminares

Antes que nada veamos las cosas que necesitamos saber para poder entender y resolver el problema.

1.1. Sudoku

El Sudoku es un juego muy popular inventado a finales de la década de 1970 que consiste en rellenar una cuadrícula de 9×9 celdas, dividida en subcuadrículas de 3×3, también llamadas "cajas", con dígitos que van desde el 1 hasta el 9, y con dígitos colocados en algunas celdas inicialmente.
Comúnmente, a una colección de nueve celdas, ya sea una fila, una columna, o una caja, se le conoce como "unidad" (*units*), y a las celdas que comparten una unidad se les conoce como "amigos" (*peers*).
Para solucionar el Sudoku, lo único que se tiene que verificar es que un dígito aparezca sólo una vez en cada unidad.
Cabe mencionar que un Sudoku está bien planteado si la solución es únnica, por lo que mi programa puede servir para indicar al usuario que tan bueno es haciendo Sudokus. El problema es que fue demostrado por el matemático Gary McGuire que no es posible tener un Sudoku bien planteado si no hay un mínimo

de 17 dígitos colocados al principio del juego [1], por lo que la interacción con el programa puede llegar a ser algo tediosa.

1.2. Estrategia

Existen muchas estrategias para resolver un Sudoku, y así como estrategias también existen muchas implementaciones. Para esta implementación decidí seguir la estrategia del problema satisfacción de restricciones (PSR).
Esta estrategia consiste en modelar el problema mediante un conjunto de restricciones sobre el espacio de las posibles soluciones que puede tener el problema y encontrar soluciones que satisfagan todas las restricciones.
La resolución de un PSR consta de dos fases:

1. **Modelar el problema como un problema de satisfacción de restricciones:** En el caso del Sudoku tenemos dos restricciones muy importantes que nos permitirán encontrar una posible solución, las cuales son

   - Si una celda sólo tiene un posible valor, entonces se eliminar ese valor de los posibles valores que se le pueden asignar a los amigos de dicha celda.
   - Si en una unidad sólo hay una posible celda para un valor, es decir, que un valor sólo pueda ser asignado a una celda de toda una unidad, entonces se coloca el valor en esa celda.

   Con estas restricciones garantizamos que el valor de las celdas que sólo contienen un valor posible no se repita en la unidad a la que pertenece, lo cual es de las cosas principales que debemos tener en cuenta.

2. **Procesar las restricciones:** Una vez formulado en problema como un PSR tenemos dos maneras de procesar las restricciones:

   - **Técnicas de consistencia:** Se basan en la eliminación de valores inconsistentes de los dominios de las variables.
   - **Algoritmos de búsqueda:** Se basan en la exploración sistemática del espacio de soluciones hasta encontrar una solución que cumpla con las restricciones.

   Para la implementación del problema usamos ambas maneras de procesar las restricciones. La primera es muy útil para reducir el espacio de búsqueda y hacer más fácil encontrar una solución. La segunda también nos es muy útil para probar las posibles soluciones y asegurar que cada dígito se encuentre al menos una vez en cada unidad.
   El problema es que no siempre es óptimo probar todas las posibles soluciones, pues algunas pueden no llevarnos a una buena solución y a hacer operaciones innecesarias de más. Es por eso que usaremos la heurística de optimización por primer fallo, o Minimum Remaining Values (MRV).

[1] Las claves matemáticas para resolver un sudoku

1.3. Optimización por primer fallo

La optimización por primer fallo es una heurística que consiste en elegir la variable con el dominio más pequeño, es decir, la variable más restringida, con el fin de que hagamos el menor número de operaciones y encontremos la solución más rápido.
En nuestro caso, después de haber aplicado la estrategia de modelar el problema como un problema de satisfacción de restricciones, vamos a fijarnos en la celda que tenga la menor cantidad de valores posibles, pues esa sería la más restringida, y una ves que tengamos esa celda probaremos con cada uno de los valores y por cada uno se formará una rama de un árbol cuya raíz será el resultado de la aplicación de la primera estrategia. Así, en algunas de las hojas de las ramas podremos tener una solución, por lo que bastaría con revisar las hojas y regresar la primera solución que encontremos.

2. Especificación

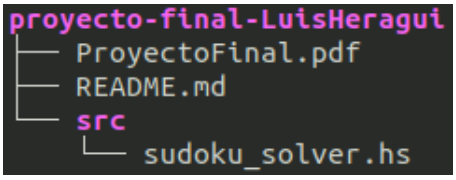
En esta sección se presenta la herramienta utilizada para la construcción del sistema, así como la implementación del mismo.

2.1. Herramientas

El sistema consta de los siguientes componentes:

- **Lenguaje de programación:** Haskell.
- **Control de versiones:** Git 2.25.1 y GitHub.

Estructura La estructura del proyecto es la siguiente:



Directamente de la raíz se encuentran los archivos *ProyectoFinal.pdf*, que es este mismo archivo, *README.md*, que contiene una breve descripción del programa, y la carpeta **src/** que contiene el archivo *sudoku_solver.hs*, el cual es la implementación del programa.

2.2. Implementación

Para resolver el problema primero se establecieron todos los datos que necesitamos para resolver un Sudoku, que son las celdas, las unidades, los amigos de cada celda, las unidades a las que pertenece cada celda, y se trabajó sobre

un tablero representado por medio de un tipo de dato llamado Array i e, el cual básicamente es un arreglo pero con la característica de que los índices no necesariamente deben ser números, entonces en este caso se usó como un arreglo de dos dimensiones el cual contiene como índice la celda del tablero, y como elemento el valor de la celda. Para evitar conversiones de un tipo de dato a otro se trabajó con Chars y Strings, y el tablero principal es un tablero en el que cada celda contiene todos los dígitos del uno al nueve.
El tablero principal representa todas las posibilidades que puede tener cada celda, y una vez que el usuario selecciona las celdas que quiere modificar e ingresa sus valores, el tablero se actualiza para tener en las celdas selecionadas los valores que ingresó el usuario.
Nos sirve tener el tablero con todos los posibles valores en cada celda porque así sólo nos tenemos que preocupar por ir eliminando elementos que ya se encuentren correctamente ubicados en una celda y no nos tenemos que fijar más que en los amigos de dicha celda.
Las funciones principales para implementar la estrategia PSR son llamadas *asigna* y *elimina*.
La función *asigna* se encarga de actualizar los posibles valores que puede tener cada una de las celdas, pero depende completamente de *elimina* porque para actualizar los valores debe de eliminar, por lo que *asigna* básicamente elimina todos los valores de una celda, menos su valor dado, y elimina se encarga de fijarse en los amigos de la celda y eliminar los valores. Esto quiere decir que *elimina* se encarga de satisfacer la primera restricción, pero también manda a llamar a una función que se encarga de satisfacer la segunda restricción cada que está eliminando un valor de las celdas amigas.
La función para satisfacer la segunda restricción se llama *evalua*, y lo que hace es evaluar un valor en las unidades en las que es posible que esté. Si en dicha unidad sólo es posible que esté en una celda, entonces se asigna el valor en esa celda y se satisface la segunda restricción.
Ahora, al terminar de aplicar las funciones anteriores es posible que aún hayan celdas con más de un posible valor, y es en donde entra la función *busca*. Esta función se encarga de construir un árbol como está descrito en la sección **1.3** y, como se mencionó, se seleccionan primero los valores de la celda que tenga menos posibles valores para que a partir de ahí se formen las ramas y sean menos caminos los que se recorran en vano.
Como es posible que no hayan soluciones, representamos al árbol como un árbol general de valores tipo (Maybe Tablero). Esto para poder descartar a las hojas de la forma Nothing y así saber cuántas soluciones hay y cuáles son, pues serían las que no sean de la forma Nothing. Y para obtener las soluciones basta con fijarnos en las hojas del árbol y agregar a una lista las que no sean de la forma Nothing. Las agregamos a una lista para poder obtener más fácil su representación en cadena y después mostrar una solución al usuario.
En cuanto a los Sudokus con más de una solución hay un problema, y es que a veces las soluciones son bastantes, por lo que decidí sólo obtener las primeras mil soluciones que se encuentren y mostrar las que el usuario quiera.

3. Manejo del sistema

Descarga del proyecto
Descargar el programa usando el siguiente comando de Git en cualquier directorio del sistema de archivos:

```
$ git clone https://github.com/ciencias-unam/proyecto-final-LuisHeragui.git
```

Se creará un directorio con la estructura definida en la sección **2.1**

Compilación y ejecución
Para esta parte se supone que el usuario tiene instalado en su sistema operativo el intérprete de Haskell llamado **GHCi**.
Entonces, una vez descargado el proyecto el usuario debe acceder a la carpeta **src/**, y una vez ahí ejecutar el siguiente comando:

```
$ ghci sudoku_solver.hs
```

Con ese comando se interpretará y compilará el programa, y el usuario se encontrará en el intérprete de Haskell con el programa compilado, por lo que basta con correr el siguiente comando para ejecutar el programa:

```
> main
```

Y eso es todo. Ya está listo para crear Sudokus y observar sus soluciones.

Referencias

[1] Wikipedia. (2021). *Sudoku* 2021, de Wikipedia en `https://es.wikipedia.org/wiki/Sudoku#cite_note-2`

[2] Fernando Sancho Caparrini. (2020). *Problemas de Satisfacción de Restricciones* 2020, de cs.us.es en `http://www.cs.us.es/~fsancho/?e=141`

[3] cornell.edu. (2011). *Constraint Satisfaction* 2020, de cornell.edu en `http://www.cs.cornell.edu/courses/cs4700/2011fa/lectures/05_CSP.pdf`