Reto 1: TDA Sudoku Killer

Adam Bourbahh Romero

25 de septiembre de 2025

1 TDA Sudoku Killer

He tratado de replicar el pensamiento humano, para ello he resuelto uno (fácil), y he observado que mi proceso se ha basado en:

1. Tratar de encontrar jaulas con una celda.
2. Aplicar la regla del 45 para conseguir la suma de la cuadrícula (por ejemplo, si una jaula está contenida por completo en una cuadrícula, sabemos que esa suma vale x, y la del resto de la cuadrícula 45-x, pues la suma en una cuadrícula 3x3 siempre es 45 (1+2+...+9)).

Dado que no soy un experto del sudoku, una vez hecho esto, he empezado a probar hasta que he encontrado de nuevo una forma de aplicar la lógica. He enfocado el TDA para poder realizar esto.

1.1 Planteamiento

Las dos grandes restricciones del Sudoku Killer son las siguientes:

- No se puede repetir número en una misma fila, columna o cuadrícula.
- Los bloques/jaulas que nos exigen un target sum.

Es por ello que he tratado de enfocar el TDA para facilitar las comprobaciones de estas exigencias, pues de otra forma podría ser un proceso muy costoso.

1.2 Especificaciones de los TDA

A nivel de especificación, y para mantener la simplicidad, recomiendo usar un struct "Posicion" que simplemente contenga una fila y una columna. A mi criterio, no creo que merezca la pena crear un TDA completo para algo tan simple.

1.2.1 TDA Celda

Este es el dato que representa cada celda del Sudoku. Le he dado los siguientes atributos:

- Un entero `valor` en el rango $[0, 9]$, donde 0 significa que la casilla está vacía.
- Un dato `Posicion` para guardar sus coordenadas.

- Algo que conecte la celda con su jaula, en este caso, un `id_jaula` de tipo entero o un puntero.
- Un dato que nos permita saber si un valor es un posible candidato. Para esto he pensado en un **array de booleanos** o quizás bitmasking.
- Un dato que nos diga en que subcuadrícula está.

1.2.2 TDA Jaula

Este dato está asociado a las jaulas y sus sumas. Lo he dotado de los siguientes atributos:

- Un entero `suma_objetivo` con el valor asociado a la suma.
- Un **contenedor de tipo** `Posicion` para guardar las coordenadas de las celdas.
- Un entero `suma_actual` para llevar la cuenta de la suma parcial.
- Un entero que diga el número de celdas sin rellenar (de forma que cuando quede solo una, el solver salte a rellenarla)

1.2.3 TDA Sudoku Killer

Esta es la parte más importante, donde todos los componentes anteriores se conectan.

- Un **array 2D de tipo** `Celda` para representar el tablero de 9x9.
- Un **contenedor de** `Jaulas`.
- Tres **arrays booleanos 2D** para las filas, columnas y cuadrículas, que sirvan para verificar posiciones en tiempo constante.

1.3 Especificación de Funciones

1.3.1 Funciones de Construcción y Acceso

`bool cargarDesdeFichero(string nombreArchivo)`

- **@brief**: Carga la configuración del tablero y las jaulas desde un fichero de texto (este método lo veo útil a la hora de hacer la clase).
- **@param nombreArchivo**: La ruta del fichero que contiene la definición del Sudoku Killer.
- **@return**: `true` si la carga fue exitosa, `false` en caso contrario.

`void colocarNumero(Posicion p, int num)`

- **@brief**: Coloca un número y actualiza las estructuras de control (arrays booleanos, suma de la jaula, etc.).
- **@param p**: La posición de la celda.
- **@param num**: El número a colocar.

Getters y Setters Esenciales: Aunque se omiten por brevedad, serían necesarias funciones para interactuar con los atributos de los TDA. Por ejemplo: `getCelda(Posicion)`, `getJaulaDeCelda(Posicion)`, `setValor(Posicion, int)`, `getSumaActual(int id_jaula)`, etc. Estas funciones permitirían al algoritmo consultar y modificar el estado del tablero de forma controlada.

1.3.2 Funciones para el Algoritmo de Resolución

A continuación especifico las funciones principales del TDA, divididas por su rol en el algoritmo.

Funciones para la Fase Lógica (Fase 1): `Contenedor<Posicion>encontrarCeldasInmediatas() const`

- **@brief**: Escanea el tablero para encontrar celdas que se pueden rellenar con certeza lógica.
- **@details**: Implementaría la búsqueda de jaulas unitarias, la aplicación del método del 45 y la búsqueda de jaulas a las que solo les falta una celda por rellenar.
- **@return**: Un contenedor con las posiciones de las celdas determinadas.

`void propagarRestricciones(Posicion p, int num)`

- **@brief**: Tras colocar un número, actualiza los candidatos de las celdas afectadas.
- **@param p**: La posición de la celda que se acaba de rellenar.
- **@param num**: El número que se ha colocado.

`Posicion encontrarMejorCeldaVacia() const`

- **@brief**: Escanea el tablero para encontrar la celda vacía con el menor número de candidatos posibles.
- **@details**: Itera sobre todas las celdas y devuelve la posición de aquella que, estando vacía, tiene el conjunto más pequeño de números posibles.
- **@return**: Un dato `Posicion` con las coordenadas de la mejor celda a rellenar.

Funciones para la Fase Recursiva: `bool esMovimientoValido(Posicion p, int num) const`

- **@brief**: Comprueba si un número viola las reglas del Sudoku clásico y si satisface la restricción de suma de su jaula.
- **@param p**: La posición de la celda a comprobar.
- **@param num**: El número candidato a comprobar.
- **@return**: `true` si el movimiento es válido, `false` si no lo es.

2 Algoritmo de Resolución Híbrido

Mi propuesta de resolución combina dos estrategias. Primero, un motor de deducción lógica para resolver todo lo posible sin adivinar, y segundo, un algoritmo recursivo que actúa como red de seguridad cuando la lógica se agota.

2.1 Fase 1: Deducción Iterativa

Esta es la fase inteligente del solver. La idea es aplicar reglas lógicas de forma repetida hasta que no se puedan deducir más casillas.

1. **Obtención de Inmediatas:** Se buscan todas las casillas que se pueden determinar con certeza usando la función `encontrarCeldasInmediatas`.
2. **Escritura y Limpieza:** Por cada casilla inmediata encontrada, se escribe en el tablero con `colocarNumero`. Cada escritura desencadena un proceso de limpieza que propaga las restricciones, usando `propagarRestricciones`.
3. **Iteración:** Se repiten los pasos 1 y 2. El bucle continúa hasta que una iteración completa no logra escribir ninguna casilla nueva.

Cuando no se puede rellenar otra casilla, pasamos a la búsqueda recursiva.

2.2 Fase 2: Búsqueda Recursiva

Esta fase se activa si la deducción lógica se detiene sin haber resuelto el puzle.

Paso 1: Se busca la casilla vacía con menos posibilidades (usando `encontrarMejorCeldaVacia`).

Paso 2: Para cada número candidato en esa casilla, se comprueba si es válido con la función `esMovimientoValido`.

Paso 3: Si el candidato es válido, se coloca con `colocarNumero` y se llama recursivamente a la función de resolución para la siguiente celda vacía. Se actualizan las restricciones con `propagarRestricciones`.

Paso 4: Si una llamada recursiva devuelve `false`, se retrocede (backtracking): se borra el número, se revierten los cambios en las estructuras de control (suma de la jaula, etc.) y se prueba el siguiente candidato. Si se agotan los candidatos, se devuelve `false`.

2.3 Aclaraciones

He tratado de ser lo más explicativo posible, obviando las funciones más simples y repetitivas. He usado el backtracking porque el proceso lógico me parecía insuficiente para considerarlo resuelto; además he valorado que es una técnica que se me podría haber ocurrido sin conocerla previamente.