Sección de Matemáticas
Universidad de La Laguna

Raúl Saavedra Hernández

Sudoku, aspectos matemáticos

Sudoku, mathematical aspects

Trabajo Fin de Grado
Grado en Matemáticas
La Laguna, Marzo de 2019

DIRIGIDO POR
Hipólito Hernández Pérez
Inmaculada Rodríguez Martín

Hipólito Hernández Pérez
Departamento de Matemáticas,
Estadística e Investigación Operativa
Universidad de La Laguna
38200 La Laguna, Tenerife

Inmaculada Rodríguez Martín
Departamento de Matemáticas,
Estadística e Investigación Operativa
Universidad de La Laguna
38200 La Laguna, Tenerife

Agradecimientos

En especial, quiero agradecer a D. Hipólito Hernández Pérez el gran esfuerzo realizado para la elaboración de este trabajo.

A mi familia, por respetar y apoyar mis decisiones tomadas.

A mis amigos, por acompañarme y ayudarme en esta etapa de mi vida.

Raúl Saavedra Hernández
La Laguna, 15 de marzo de 2019

Resumen · Abstract

Resumen

En el presente trabajo se estudian los aspectos matemáticos del Sudoku. Comenzamos introduciendo las reglas del juego y observando su evolución a lo largo de la historia, desde su origen hasta la actualidad. Luego, estudiamos diferentes métodos de resolución, entre los que destacamos el Backtracking y los métodos de resolución por humanos. Finalmente, elaboramos un algoritmo, aplicando estos métodos, que resuelve y calcula todas las posibles soluciones, genera sudokus de forma aleatoria y evalúa la dificultad.

Palabras clave: *Sudoku – Backtracking – Optimización Combinatoria – Matemática Discreta.*

Abstract

The mathematical aspects of Sudoku are studied in this project. First, we introduce the rules of the game and we observe its evolution throughout history, from its origin to the present. After, we study different resolution methods, among which we stand out the Backtracking and the methods of resolution by humans. Finally, we elaborate an algorithm, applying these methods, that solves and calculates all possible solutions, generates sudokus randomly and evaluates the difficulty.

Keywords: *Sudoku – Backtracking – Combinatorial Optimization – Discrete Mathematics.*

Contenido

Introducción

El Sudoku es un pasatiempo que consiste en rellenar una cuadrícula de $N \times N$ con N símbolos, sin repetir ninguno de ellos en una fila, columna o región.

El origen de este juego es, posiblemente, el cuadrado de Euler, cuya única diferencia es la regla de no repetir los símbolos en las regiones. Otra teoría afirma que fue creado en Nueva York, a finales de 1970, por la revista "Math Puzles and Logic Problems". Posteriormente, se hizo famoso en otros países alcanzando la repercusión actual.

Una cuadrícula de $N \times N$ en blanco, con $N = 1$ tiene 1 solución, para $N = 4$ obtenemos 288 soluciones. Al aumentar a $N = 9$, el número de soluciones alcanza la cifra de 6.670.903.752.021.072.936.960, cuya descomposición en números primos nos confirma que, contarlas, no es un simple problema de combinatoria.

Hemos dedicado gran parte del trabajo a la realización de un algoritmo que resuelve los sudokus mediante *Backtracking*, a la vez que cuenta todas las posibles soluciones. Además, los genera de forma que solo tengan una única solución. Por último, evalúa la dificultad de un sudoku, dependiendo de los métodos de resolución por humanos necesarios para alcanzar la solución final.

Comenzaremos describiendo el juego, estudiando su evolución a lo largo de la historia y explicando algunas estrategias para resolver el sudoku a mano en el capítulo 1.

Luego, estudiaremos, en el capítulo 2, algunos métodos de resolución como el *Backtracking* y la programación lineal entre otros, más orientados a la aplicación de la informática para resolverlos.

Por último, explicaremos como hemos implementado el algoritmo que hemos elaborado para resolver, generar y evaluar la dificultad de los sudokus, en el capítulo 3.

1

Introducción al Sudoku

En este capítulo comenzaremos describiendo el juego y observando algunos detalles curiosos sobre el mismo. Repasaremos su origen y desarrollo a través de la historia y finalizaremos con las diferentes estrategias de resolución aplicadas por los humanos.

1.1. Descripción del juego

En la actualidad, el sudoku, es uno de los pasatiempos más famosos del mundo. Gran parte de las revistas y periódicos de prácticamente todo el mundo tienen un hueco reservado en sus páginas de entretenimiento para este juego. La fácil comprensión de las reglas hace a este pasatiempo atractivo para que personas de todas las edades se atrevan a intentar resolverlo, aunque esto no implica que sea simple de solucionar.

El juego consiste en rellenar una tabla de 9×9, 81 casillas o celdas en total, dividida en regiones de 3×3, con las cifras del 1 al 9. Inicialmente, algunos números vienen fijados en algunas de las casillas, denominados pistas. La condición esencial del juego se basa en escribir una sola vez cada cifra en cada grupo (columna, fila y región). Desde el punto de vista de la optimización, el sudoku, presenta el fenómeno de explosión combinatorial, puesto que, las dimensiones y complejidad matemática que resulta resolver el problema es grande. Por ello, está clasificado como un problema NP-completo, es decir, no existe un algoritmo que lo resuelva en un tiempo de cómputo polinomial en el tamaño N del problema.

La cuadrícula de 9×9 con regiones de 3×3 es la más común. También existen otros tamaños, generalmente, de dimensión $N \times N$ con regiones de $\sqrt{N} \times \sqrt{N}$, en este caso, se utilizan N números o caracteres para llenar la cuadrícula, como 4×4, 16×16, 25×25. También podemos encontrar cuadrículas con regiones que no son cuadradas.

Generalmente, se dice que, un sudoku está bien planteado cuando solo tiene una única solución, para ello, demostrado por McGuire, Tugemann y Civario (2013) [6], el número mínimo de pistas necesario es 17. Esto no implica que todo sudoku con 17 datos iniciales, o más, esté bien planteado.

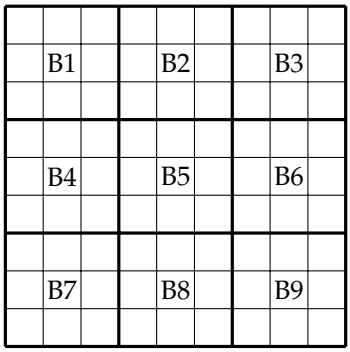
Cuando nos disponemos a enfrentarnos a un sudoku, podemos encontrar varios niveles de dificultad. Esta dificultad depende de la relevancia y la posición de los números dados, no de la cantidad de pistas iniciales, puesto que, un jugador ante una cuadrícula con un número de pistas bajo, lo que él vería como una cuadricula casi vacía, podría resultarle más fácil de resolver que uno con un número de pistas iniciales mayor. La dificultad reside en los métodos de resolución que tenga que utilizar el jugador para rellenar la cuadrícula, ante métodos de mayor elaboración se considera un nivel de dificultad más alto.

Si empezamos con la cuadrícula en blanco, es decir, ninguna pista inicial, podemos observar que para un sudoku de orden 1 la cantidad de soluciones es 1, mientras que cuando el orden es 4 hay 288 soluciones. Usando combinatoria, vemos que, tenemos 4! formas de llenar la primera fila. En cuanto a la segunda fila, una vez fijada la primera fila, hay 2 opciones para las casillas que pertenecen a la primera región y otras 2 para las casillas de la segunda región, en total 4 formas de rellenarla. En la tercera fila, una vez fijadas las dos anteriores, observamos que, dependiendo de la fila dos, puede haber 4 o 2 formas de rellenarla, por tanto, la media de opciones para la fila 3 es 3. La cuarta fila, una vez fijadas las otras tres, solo tiene una opción, con lo que concluimos que $4! \cdot 4 \cdot 3 = 288$.

En el caso de un sudoku de orden 9, se ha estimado que el número de soluciones es 6.670.903.752.021.072.936.960. Su descomposición en factores primos es $2^{20} \cdot 3^8 \cdot 5 \cdot 7 \cdot 27.704.267.971$. Un número primo tan grande como el 27.704.267.971 implica que no podemos contar el número de soluciones como un simple problema de combinatoria, para conseguirlo ha hecho falta la computación.

1.2. Número de soluciones

Felgenhauer y Jarvis (2005) [4] explican como alcanzar las 6.670.903.752.0 21.072.936.960 soluciones utilizando un programa informático que consiste en reducir dicha cantidad mediante variaciones que dan soluciones equivalentes.



Etiquetando las regiones como vemos en la cuadrícula anterior, comienzan rellenando B1 de forma canónica, es decir, los números del 1 al 9 de forma ordenada reduciendo así la cantidad de soluciones en 9!.

Continúan rellenando B2 y B3 de forma pura, es decir, colocando en las filas de los bloques los subconjuntos $\{1,2,3\}$, $\{4,5,6\}$ $\{7,8,9\}$, con $(3!)^6$ formas diferentes, y de forma mixta, como puede ser ejemplo $\{4,5,7\}$, $\{8,9,1\}$ $\{6,2,3\}$, en las que intercambiando los números 1, 2 y 3 encontramos $3 \cdot (3!)^6$ maneras diferentes. Logrando así reducir la cantidad en 2.612.736 soluciones, puesto que, tenemos 2 posibles filas superiores puras, y 18 filas superiores mixtas.

Luego, generan una reducción lexicográfica tomando todas las 2.612.736 posibilidades mencionadas anteriormente. Primero, catalogando las permutaciones de las columnas dentro de B2 y B3 para que las primeras entradas estén en aumento. Segundo, intercambiando B2 y B3, si es necesario, para que B2 aparezca antes que B3 en un orden lexicográfico. Así, reduciendo el número de posibilidades que debemos considerar a 36.288.

Una vez hecho público el método, compararon resultados con otro programa, ahora realizado por Ed Russell, que resultó ser más rápido pero se obtuvieron los mismos resultados.

1.3. Historia

El origen de este juego tiene varias teorías que veremos y desarrollaremos a continuación.

Una de las teorías más aceptadas sobre la procedencia del sudoku sostiene que en el siglo XVIII Leonard Euler (1707-1783) fue quien creo este juego, indirectamente, al establecer las pautas para el cálculo de probabilidades con el objetivo de representar una serie de números sin repetir. De hecho, Euler llegó a escribir los cuadrados latinos. Un cuadrado de Euler o cuadrado latino ortogonal de orden N consisten en un cuadrado de $N \times N$ casillas donde situamos

N elementos diferentes, sin que los elementos se repitan en cada fila ni en cada columna. Observamos así que, cada solución de un sudoku es un cuadrado de Euler pero el reciproco, en general, no es cierto, puesto que, el sudoku tiene la restricción añadida de no repetir elemento en las regiones.

Por otra parte, en Nueva York a finales de los años 1970, la revista "Math Puzles and Logic Problems", especializada en rompecabezas matemáticos y problemas lógicos, asociada a la empresa editora Dell Magazines creó el juego "Number place", considerado la primera versión del sudoku. Se desconoce el nombre del diseñador de este juego aunque muchos se lo asocian a Walter Mackey, uno de los diseñadores de puzles de Dell. De todas formas, con el paso del tiempo el juego cayó en el olvido.

Años más tarde, una empresa editora japonesa especializada en pasatiempos para la prensa y revistas, llamada Nikoli, trasladó este juego a Japón, publicándolo en el periódico "Monthly Nikolist" en el año 1984 con "Sūji wa dukoshin ni kagiru" como título, cuya traducción es "los números deben estar solos". Kaji Maki, director de la empresa, fue quien le puso el nombre que posteriormente sería abreviado hasta su nombre actual sūdoku, donde "sū" es número y "doku" es solo, es decir, número solo.

Dos años después, en 1986, Nikoli aumentó la popularidad del sudoku al añadirle dos innovaciones. Primero, puso como restricción que el sudoku tendría un máximo de 30 pistas. Además, como segunda innovación, los sudokus serían "rotacionalmente simétricos", es decir, las casillas con las pistas estarían dispuestas simétricamente, sin embargo, en los sudokus actuales no se suele observar esta característica. Asimismo, con algunas variaciones más, el sudoku se fue extendiendo por la prensa japonesa.

El sudoku llegó a los ordenadores en 1989, con la primera versión cuyo nombre fue DigitHunt, registrada por la empresa Loadstar Softdisk Publishing y publicada en la revista Commodore 64. Posteriormente, Wayne Gould, un abogado nacido en Nueva Zelanda que se trasladó a trabajar en 1982 a la ciudad de Hong Kong, llegando a ser juez de la corte en esta ciudad, se encontró con una revista de sudokus durante unas vacaciones en Japón. Gould, motivado por la popularidad del juego en los habitantes de Japón, intentó difundir el juego por otros países, principalmente en Gran Bretaña. En 1997, tras jubilarse, pasó 6 años elaborando un programa de ordenador que generaba una gran cantidad de sudokus en poco tiempo. Esto hizo que los sudokus apareciesen en periódicos de toda Europa y América.

Gould, en busca de su objetivo, le envió varios sudokus al periódico londinense "The Times" para que los publicara. Esta publicación tuvo que esperar hasta el 12 de noviembre de 2004, puesto que, el periódico tardó en atender la oferta. Tres días después, el periódico "The Daily Mail" copió el juego y lo publicó bajo el nombre de "Code Number". Este movimiento continuó, hasta que, prácticamente toda la prensa británica publicó el juego. Por otra parte, Gould

se dedicó a escribir libros sobre sudokus y fue editor literario de una colección de obras sobre este juego.

Otra empresa editora llamada Kappa, dedicada a los pasatiempos, recuperó todos los sudokus de Nikoli y los publicó en la revista "Games Megazine", dándole el nombre de "Square Hawaii". Actualmente, hay gran cantidad de publicaciones dedicadas únicamente al sudoku, además de que la mayoría de los periódicos de todos los países publican este juego en sus páginas de entretenimiento. Hoy en día, Dell Magazines, la compañía pionera en publicar sudokus, sigue ofertando diferentes revistas dedicadas a ellos.

En el verano de 2005, "Sky One", un canal británico de televisión, emitía por primera vez un programa dedicado al sudoku. En él, nueve equipos de nueve jugadores intentaban resolver sus respectivos sudokus, donde se permitía la interacción del público. El programa no tuvo el éxito esperado, quizás, por la dificultad que lleva adaptar este juego a una emisión televisiva. Por otro lado, ese mismo año, en una colina cerca de Bristol en Inglaterra, se logró hacer el sudoku mas grande del mundo con 84 metros de largo.

Actualmente, podemos encontrar una infinidad de paginas web que nos facilitan, gratuitamente, sudokus de diferentes dificultades y tamaños donde puedes comprobar si cometes un error, o conocer la solución, al instante. Además, podemos descargar una gran variedad de aplicaciones para los móviles con las mismas prestaciones que las páginas web.

1.4. Métodos de resolución por humanos

En esta sección vamos a describir algunos métodos, en orden de menor a mayor dificultad, con los que podremos solucionar, prácticamente, cualquier sudoku que nos podamos encontrar.

En primer lugar, debemos encontrar los candidatos, es decir, aquellas posibles cifras que podrían ubicarse en las casillas que vienen vacías o, lo que es lo mismo, sin pista.

1.4.1. Ubicación individual de candidatos

Una vez conocidos los candidatos de cada casilla, podemos iniciar la colocación de algunos números con certeza.

1. **Único desnudo.**
   Decimos que tenemos un único desnudo cuando para una casilla solamente hay un candidato.
   En la figura 1 podemos observar un único desnudo, el 4, en la casilla marcada con un tono amarillo. Posteriormente, se eliminaría esta cifra de los candidatos en las celdas marcadas en gris, que corresponden a los grupos

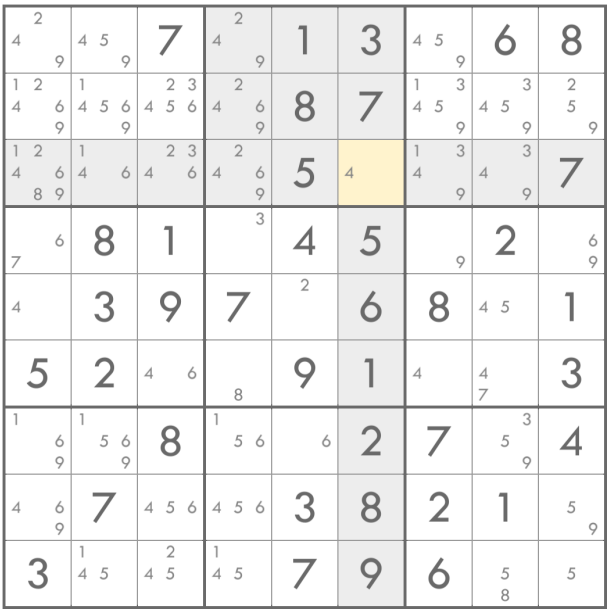


Figura 1.1. Ejemplo de único desnudo

que pertenece la casilla.

2. **Único oculto.**
   Llamamos único oculto a un candidato que, dentro de una fila, columna o región, solo se encuentra en una casilla. Recibe el nombre de oculto porque dentro de esta celda hay mas candidatos pero, al ser la única donde se puede colocar dicho valor dentro de un grupo, ese candidato se le asigna a esa casilla con seguridad.
   Observamos un único oculto en la figura 2, el número 3 solo se puede colocar en la casilla destacada dentro de la fila y región marcadas.

3. **Intersección Línea-Región.**
   Los métodos anteriores nos permiten asignar los valores a las casillas con total certeza, llegado el momento en que no es posible seguir colocando más números debemos comenzar a eliminar candidatos, para ello, pasamos a desarrollar métodos de eliminación de candidatos.
   La intersección Línea-Región se puede dar de dos formas, Fila-Región o Columna-Región. Supongamos que tenemos un valor que debe, obligatoriamente, ser colocado en una región, además observamos que solo se puede colocar en una fila de esta región (análogamente una columna), dicho

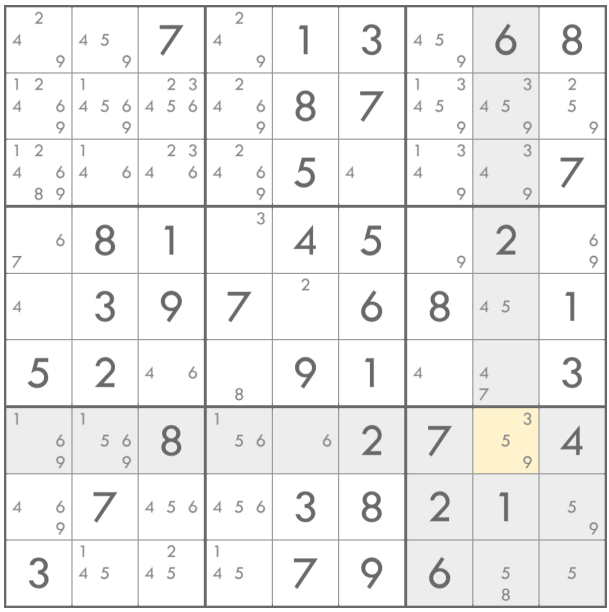


Figura 1.2. Ejemplo de único oculto

valor puede ser eliminado del resto de la fila (análogamente de la columna) que no pertenezca a la región. Entonces diremos que tenemos una intersección Fila-Región (análogamente una intersección Columna-Región).

1.4.2. Ubicación grupal de candidatos

A partir de ahora, comenzamos a trabajar con más de un candidato con el objetivo de seguir eliminando posibles valores de las casillas vacías. Para ello, definiremos subconjunto desnudo, subconjunto oculto y veremos los métodos donde se aplican.

1. **Subconjunto desnudo.**

   Definición 1.1. *Un subconjunto es un conjunto de casillas de un mismo grupo en el que coincide el número de candidatos con el número de celdas del conjunto. Asimismo, un subconjunto es un subconjunto desnudo si todas las casillas que lo forman tienen los mismos candidatos.*

   - Par desnudo.
     En este caso, tenemos dos casillas de un mismo grupo con el mismo par de candidatos, solo esos dos posibles valores. Entonces, podemos borrar estos dos candidatos de las demás celdas del grupo.

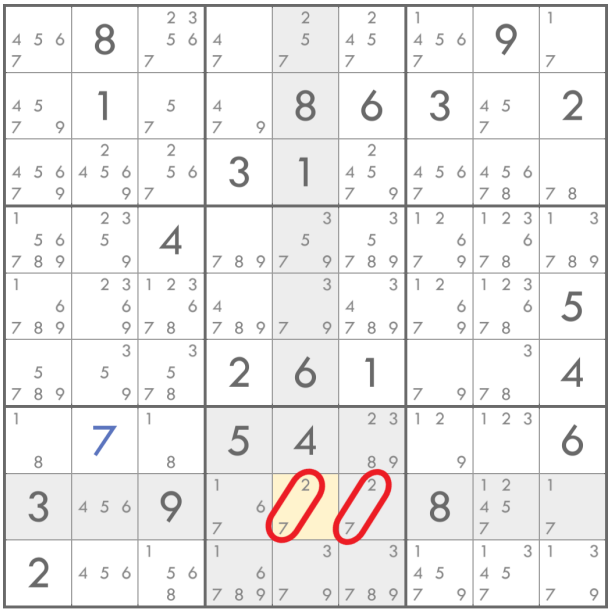


Figura 1.3. Ejemplo de par desnudo

Como ejemplo de par desnudo, observamos en la figura 1, las cifras 7 y 2 constituyen un par desnudo de la fila y región marcada.

- Trío desnudo.
  Cuando observamos tres casillas de un grupo con los mismos tres candidatos procedemos, como con el par desnudo, a eliminar los candidatos de las celdas restantes del grupo. Sin embargo, matizamos que no es necesario que estén los tres candidatos en las tres casillas, si no que se puede dar diferentes combinaciones de dos de ellos.

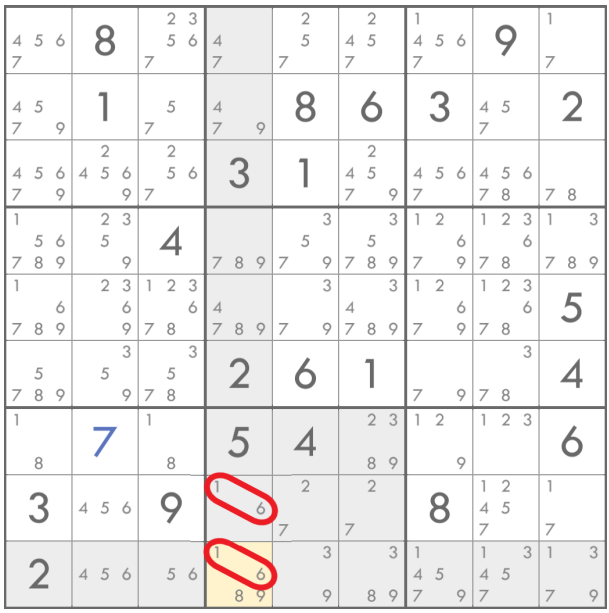
De manera similar, podríamos encontrarnos con un cuarteto desnudo y llevar a cabo la eliminación de los respectivos candidatos de las celdas del grupo que no pertenecen a este subconjunto desnudo.

2. **Subconjunto oculto**

Definición 1.2. *Un subconjunto oculto es un subconjunto de casillas de un grupo cuyas celdas contienen a los candidatos del subconjunto y otros posibles valores, sin embargo, los candidatos del subconjunto no aparecen en ninguna otra celda del grupo.*

- Par oculto

Cuando en un grupo encontramos dos casillas que tienen dos candidatos iguales y estos no aparecen en otra celda del grupo, podemos eliminar con certeza los demás candidatos de estas dos casillas.



En la figura 2 podemos observar un par oculto formado por las cifras 1 y 6. En la casilla marcada en amarillo podríamos eliminar los números 8 y 9.

- Trío oculto
  De forma análoga, si observamos tres casillas que tienen tres candidatos iguales y estos no aparecen en otra celda del grupo, los otros candidatos de estas tres casillas pueden ser eliminados. Como sucedía con el trío desnudo, los tres candidatos no tienen porque aparecer en las tres celdas, es posible que haya una combinación de dos de ellos.

2

Metodos de resolución

Dedicaremos el siguiente capítulo a la explicación de diferentes métodos o estrategias de resolución dentro del ámbito computacional. Trataremos sus planteamientos, el desarrollo y el éxito de finalización.

2.1. *Backtraking*

La estrategia de vuelta atrás, *backtraking*, encuentra soluciones a problemas que satisfacen restricciones. El algoritmo hace una búsqueda en profundidad durante la cual, si encuentra una alternativa incorrecta, retrocede al paso anterior y toma otra alternativa. Se suele implementar de manera recursiva.

Basado en el método de ensayo y error, el *backtraking* es un algoritmo que localiza la primera casilla en blanco del sudoku, coloca el número 1, comprueba si no existe conflicto, es decir, que no haya otro 1 en la misma fila, columna o región. En el caso de encontrar conflicto, le suma uno a la casilla, por tanto pondría el 2 y haría el mismo proceso. Si no hay conflicto, avanza hasta la siguiente casilla en blanco y repite el proceso.

Una vez avanzado el proceso puede ocurrir que llegue a una casilla en blanco, pruebe los números del 1 al 9 y no sea posible colocar ninguno. En tal caso, vuelve a la casilla que rellenó anteriormente, le suma uno y hace las comprobaciones pertinentes para poder seguir avanzando. El procedimiento retrocede las veces que sea necesario.

Este método explora exhaustivamente todas las soluciones posibles y, por tanto, termina encontrando la solución, si existe. Puede suceder que nos encontremos con un sudoku mal planteado, es decir, que no tiene una única solución, entonces, el *backtraking*, encuentra las múltiples soluciones que pueda tener.

2.2. Programación Lineal

El sudoku se puede plantear como un problema de programación lineal. No hay una función objetivo que maximizar o minimizar pero podemos resolverlo, únicamente, aplicando restricciones a las variables de decisión.

Definimos los siguientes conjuntos para la modelización del problema:

- $I = \{1, ..., 9\}$ es el conjunto de filas.
- $J = \{1, ..., 9\}$ es el conjunto de columnas.
- $K = \{1, ..., 9\}$ es el conjunto de posibles cifras que podemos colocar en cada casilla.
- $H = \{1, 4, 7\}$ son los valores de las filas y columnas en las casillas iniciales de una región.
- S es el conjunto de casillas que vienen rellenadas, con las denominadas pistas. Sus elementos son los pares ordenados (i, j).

Introducimos una variable x_{ijk} definida de la siguiente forma:

- $$x_{ijk} = \begin{cases} 1, \text{ si la cifra } k \text{ se coloca en la casilla } (i,j) \\ \\ 0, \text{ si el valor } k \text{ no se coloca en la casilla } (i,j) \end{cases}$$

El modelo matemático sería:

$$min \quad 0 \tag{2.1}$$

Sujeto a:

$$\sum_{k \in K} x_{ijk} = 1 \qquad \forall i \in I, \forall j \in J \tag{2.2}$$

$$\sum_{i \in I} x_{ijk} = 1 \qquad \forall j \in J, \forall k \in K \tag{2.3}$$

$$\sum_{j \in J} x_{ijk} = 1 \qquad \forall i \in I, \forall k \in K \tag{2.4}$$

$$\sum_{i=h_0}^{h_0+2} \sum_{j=h_1}^{h_1+2} x_{ijk} = 1 \qquad \forall h_0, h_1 \in H, \forall k \in K \tag{2.5}$$

$$x_{ijk} = 1 \qquad \forall (i,j) \in S,\ k \text{ valor de la casilla} \tag{2.6}$$

$$x_{ijk} \in \{0, 1\} \qquad \forall i \in I, \forall j \in J, \forall k \in K \tag{2.7}$$

Con las restricciones (2.2) nos aseguramos de que en cada casilla solo haya un número. Para que cada cifra aparezca una única vez en cada fila tenemos las restricciones (2.3). De forma similar las restricciones (2.4) hace que cada cifra

solo aparezca una vez por columna. En las restricciones (2.5) nos encargamos de que no se repitan las cifras en la región. Hacemos que las variables tomen el valor 1, directamente, cuando la casilla contiene una de las pistas iniciales en las restricciones (2.6). Por ultimo, con (2.7) nos aseguramos que las variables tomen los valores 1 o 0. No distinguimos si la casilla contiene pista o está vacía debido a que no es necesario.

2.2.1. Implementación del modelo matemático

Implementaremos el modelo matemático en Gusek, un programa informático que facilita un entorno de código abierto donde se puede programar y resolver los distintos problemas de programación lineal que se le puedan plantear. Tiene una versión personalizada del editor SciTE vinculado al solucionador autónomo GLPK.

Encuentra e imprime, de forma instantánea, la solución del sudoku. Ante un sudoku con más de una solución, al encontrar una de ellas, no continúa la búsqueda de soluciones. También nos informa de las variables y memoria utilizada.

A continuación, mostraremos como traducir el problema de programación lineal del sudoku para que Gusek lo resuelva:

```
param givens{1..9, 1..9}, integer, >= 0, <= 9, default 0;
/* the "givens" */

var x{i in 1..9, j in 1..9, k in 1..9}, binary ;
/* x[i,j,k] = 1 means cell [i,j] is assigned number k */

s.t. fa{i in 1..9, j in 1..9, k in 1..9: givens[i,j] != 0}:
    x[i,j,k] = (if givens[i,j] = k then 1 else 0);
/* assign pre-defined numbers using the "givens" */

s.t. fb{i in 1..9, j in 1..9}: sum{k in 1..9} x[i,j,k] = 1;
/* each cell must be assigned exactly one number */

s.t. fc{i in 1..9, k in 1..9}: sum{j in 1..9} x[i,j,k] = 1;
/* cells in the same row must be assigned distinct numbers */

s.t. fd{j in 1..9, k in 1..9}: sum{i in 1..9} x[i,j,k] = 1;
/* cells in the same column must be assigned distinct numbers */

s.t. fe{I in 1..9 by 3, J in 1..9 by 3, k in 1..9}:
    sum{i in I..I+2, j in J..J+2} x[i,j,k] = 1;
/* cells in the same region must be assigned distinct numbers */
```

Con el parámetro *given* definimos una matriz cuyos valores deben ser enteros entre 0 y 9. Cuando toma el valor 0 consideramos que la casilla la tiene que rellenar el problema y en otro caso que es un número ya dado (pista). Por

defecto, los inicializamos a 0. Seguidamente, definimos las variables teniendo en cuenta que toman valores 0 o 1.

Posteriormente, escribimos las diferentes restricciones. Le asignamos el valor 1 a las variables correspondientes cuando la casilla contiene una pista inicial. A continuación, escribimos la restricción (2.2) donde asignamos una cifra a cada casilla, la (2.3) para evitar que se repitan las cifras en cada fila y, de forma similar, la restricción (2.4) donde evitamos que se repitan los números en cada columna. Por último, escribimos la restricción que evita repetir las cifras en cada región

La forma de introducir el sudoku en Gusek es mediante una matriz donde colocamos el número correspondiente a la pista y un punto en el caso de las casillas en blanco, también puede leerlo desde ficheros Excel.

Tarda, aproximadamente, un segundo en ejecutarse y utiliza cerca de 1 Mb de memoria.

2.3. Otros métodos

Esta sección vamos dedicarla a una breve explicación de diferentes métodos, cuya elaboración resulta más compleja, como son la búsqueda tabú, algoritmos genéticos y la coloración de grafos, pudiendo encontrar información más detallada en el articulo escrito por Franco, Gómez y Gallego (2007) [5]. Puesto que, el problema en cuestión pertenece a la categoría de problemas NP-completos estos métodos se consideran poco eficientes, ya que, es necesario una gran cantidad de tiempo y no aseguran encontrar una solución.

2.3.1. Búsqueda Tabú

La Búsqueda Tabú es una técnica de optimización combinatorial proveniente de la inteligencia artificial y utiliza conceptos de memoria adaptativa y exploración sensible. Resuelve problemas del tipo:

$$\begin{aligned} &\text{Min } f(x) \\ &s.a.\ x \in X \end{aligned} \tag{2.8}$$

donde f es una función, lineal o no lineal, y X es un conjunto de restricciones lineales o no lineales. La naturaleza de la variable x puede ser continua, entera o mixta.

Cuando tenemos una mala decisión que proviene de una estrategia nos resulta más beneficioso que un acierto proveniente de una selección aleatoria, puesto que, nos da más información. En esto se basa la exploración sensible de la búsqueda tabú, lleva a cabo un método de búsqueda local para explorar eficientemente el espacio que rodea una configuración.

Conocemos por vecindad al conjunto de soluciones vecinas asociado a cada solución y entendemos por configuraciones el espacio de soluciones del problema. La búsqueda local utiliza la vecindad para pasar a la mejor configuración vecina, o si nos encontramos en la mejor, pasamos a la menos peor, donde aplicamos otra vez la búsqueda local con el objetivo de alcanzar la configuración óptima. Este método requiere un alto costo computacional, esto se debe a que en cada paso debemos analizar todas las configuraciones vecinas o un conjunto de ellas cuyo tamaño sigue considerándose grande.

A la hora de aplicar esta estrategia al sudoku, debemos comenzar rellenando los espacios libres de la cuadrícula, de tal forma que, completamos las columnas con las cifras que faltan. De esta manera, tenemos que, en las columnas no se repite ninguna cifra, pero, generalmente, esto no sucede en las filas y regiones.

Las configuraciones vecinas están definidas por las siguientes estrategias:

- Cambiar un número repetido en una fila por uno que falte en ella.
- Sustituir una cifra repetida en una región por otra que no aparezca en ella.
- Cambiar un número repetido en una columna por otro que falte en la columna. (Esta estrategia tiene sentido una vez avanzado el método, puesto que, inicialmente, no se repiten las cifras en las columnas).
- Permutar las cifras de dos casillas de una misma fila.
- Intercambiar las cifras de dos casillas de una misma columna.

Realizando una estimación de la variación de la función objetivo en el paso a una configuración vecina disminuye considerablemente el tiempo computacional, debido a que no tenemos que calificar la función objetivo recorriendo la cuadrícula y analizado el número de repeticiones por filas, columnas y regiones.

2.3.2. Algoritmos genéticos

El método de algoritmos genéticos utiliza los mecanismos de la evolución y de la genética natural para buscar en el espacio de soluciones.

Iniciamos el método seleccionando un conjunto de configuraciones iniciales, la población inicial. A continuación, realizamos un proceso de selección entre la población inicial, donde asignamos el derecho a intervenir en la generación de nuevos descendientes, seguidamente, formamos parejas que sometemos a un proceso de recombinación y finalizamos realizando mutaciones en algunas configuraciones. Este proceso pretende diversificar la población bajo el control de la tasa de mutación, un parámetro que establece cuantos cambios se realizan. Una vez finalizado el proceso obtenemos elementos de una nueva generación. Tener éxito aplicando este método depende, mayormente, de una

buena elección del tamaño de la población y de la tasa de mutación, asimismo, de la forma en que se realizan los procesos de selección, recombinación y mutación.

Veamos como se aplica este método a la resolución del sudoku, para ello, consideramos las siguientes etapas:

1. **Codificación del problema:**
   Las configuraciones son matrices de $N \times N$ con las cifras entre el 1 y N
2. **Población inicial:**
   Generamos de forma aleatoria la población inicial. Para cada configuración de este conjunto, rellenamos las casillas en blanco con un número aleatorio entre el 1 y N.
3. **Calculo de la función objetivo:**
   La función objetivo debe mostrarnos la suma del recuento de cifras que no aparecen en las filas, columnas y regiones.
4. **Proceso de selección:**
   El proceso de selección es proporcional y se realiza mediante el método de la ruleta, para ello, utilizamos una función de adaptación que garantiza selectividad y nos cambia la finalidad de minimizar a maximizar de la función objetivo inicial.
   $$Fa_i = max(\overline{Fo}) \cdot k - Fo_i \tag{2.9}$$
   donde:
   - Fa_i es la función de adaptación del individuo i-ésimo
   - $\overline{Fo}$ es el vector que almacena el valor de la función objetivo de las configuraciones.
   - k es la tasa de adaptación, tiene un valor mayor que 1.
   - Fo_i es el valor de la funcion objetivo del individuo i-ésimo.
5. **Proceso de recombinación:**
   Este proceso consiste en utilizar una pareja de configuraciones como padres que den lugar a dos hijos, donde el primero mantiene las mejores regiones de los padres, aquellos que contienen menos cifras repetidas, mientras, el segundo, conserva las mejores filas o columnas de los padres, cuya selección, completamente aleatoria, tiene la misma probabilidad.
6. **proceso de mutación:**
   Con la intención de hacer una buena exploración de búsqueda en el espacio de las soluciones y generar diversidad aplicamos este proceso en el que realizamos las siguientes cuatro mutaciones diferentes:
   - Permutamos dos cifras pertenecientes a una misma fila.
   - Permutamos dos cifras pertenecientes a la misma columna.
   - Permutamos dos cifras pertenecientes a una misma región.
   - Cambiamos un número de la cuadrícula por otro generado de forma aleatoria entre el 1 y N

Por otra parte, puede suceder que generemos una población donde ninguno de los individuos sea mejor que los considerados sus padres. En este caso, remplazamos el peor individuo de la población por el mejor de la generada. El algoritmo genético repetirá el proceso hasta que encuentre una solución o cuando se alcance el número de iteraciones predefinido.

2.3.3. Coloración de grafo

Una forma de resolver el sudoku podría ser equiparando la tarea a un problema de coloración de grafos, en el que dos celdas adyacentes no pueden tener el mismo color. En nuestro caso, contamos con 9 colores diferentes y 81 vértices, de los cuales alguno ya está coloreado.

El problema de coloración es bastante complejo, puesto que cada cuadrícula de 9×9 tiene cientos de bordes. Cada celda está contenida en una columna que incluye 8 casillas más, en una fila con otras 8 celdas y en una región con 8 casillas, de las cuales 4 ya las hemos contado entre la fila y la columna, es decir, cada una de las 81 celdas están conectadas a otras 20. Esto hace que haya un total de 1.620 aristas incidentes en alguna casilla que si dividimos por dos hacen un total de 810 aristas.

3

Programa que resuelve, genera y evalúa la dificultad

Este último capítulo describe una herramienta informática que hemos elaborado para la resolución, generación y cálculo de dificultad de sudokus. Realizado con el objetivo de aprender y profundizar en la programación destinada a la resolución de sudokus. Eligiendo *C++* como lenguaje de programación y trabajando con *Microsoft Visual Studio* como entorno de programación. Esta herramienta resuelve sudokus aplicando *Backtracking* y los métodos de resolución por humanos. Genera sudokus de forma aleatoria, intentando conseguir que tengan solución única y, por último, evalúa la dificultad según cuanta dedicación le supondría a un humano.

3.1. Estructura de datos

Para la realización de este programa hemos creado una clase a la que le hemos dado el nombre *Sudoku*. Hemos definido dentro de la clase las funciones privadas necesarias para realizar los diferentes procesos que explicaremos en las siguientes secciones. Por otra parte, hemos definido, también privado, una matriz bidimensional de 9×9, tipo *integer*, que hemos llamado *data[i][j]*, donde i se refiere a la fila y j a la columna. En ella almacenaremos el valor de la casilla, siendo 0 si está vacía. Asimismo, hemos definido una matriz tridimensional de $9 \times 9 \times 9$, tipo *bool*, llamada *candidates[i][j][k]* donde i y j siguen siendo fila y columna, respectivamente, mientras que k es la cifra. En ella almacenaremos *true* si la cifra k es candidata a ocupar la casilla (i, j) y *false* en caso contrario. Por otro lado, hemos declarado como públicas las funciones que llamamos desde la función principal *main*. Son las siguientes:

- La encargada de abrir un archivo y colocar las cifras de las celdas en la matriz *data*, *read_inputfile*.
- Para comprobar que no hay errores tenemos *check* y *check_box*.

- Llamamos a los procesos mediante *execute_backtracking*, *ramdon_generator* y *difficulty_calculator*.
- Luego, con *write* imprimimos el sudoku por pantalla.
- Además, a la hora de calcular la dificultad, *write_table_position*, nos imprime por pantalla el sudoku de una manera más dinámica, mostrándonos las casillas fijadas y los posibles candidatos de las celdas vacías.

3.2. Número de soluciones (*Backtraking*)

En primera instancia, tras comprobar que no hay problema al leer el archivo, contrastamos que el sudoku no contiene errores. Par ello tenemos la función *check* que recorre la matriz *datos* y al encontrar un número fijado como pista recurre a la función *check_box*. Esta, a su vez, aplica las funciones *check_box_in_row*, *check_box_in_colum* y *check_box_in_region*, encargadas de examinar que no se repita el número en los distintos grupos a los que pertenece la casilla.

Dado el caso en que no se repite el número, cada una de ellas irá devolviendo verdadero y el proceso continuará hasta comprobar la matriz entera. En caso contrario, si el número se repite en algún grupo, la función encargada de ese grupo devolverá falso llegando a *check* que cortará el proceso, devolverá falso y nos saltará por pantalla el mensaje de que el sudoku no tiene solución posible.

Una vez comprobado que el sudoku no contiene errores pasamos a resolverlo mediante el *backtracking* con la función *execute_backtracking*. En ella se acude a la función *fill_a_position*, pasandole la posición y el número de soluciones, que inicialmente ambas son 0. Comienza comprobando si en la posición que se encuentra hay un número fijado, en tal caso se llama recursivamente avanzando a la siguiente posición. Sin embargo, si nos encontramos en una casilla vacía, estudia mediante un bucle *for* el primer número que se podría escribir en esa casilla, para distinguir el candidato utiliza *check_box*. Una vez encontrado lo fija en la matriz *data* y avanza una posición. Si sucede que llegados a este punto no encontramos ningún candidato, lo fijamos a cero, vuelve a la posición anterior y busca el siguiente candidato. Cuando llegamos a rellenar toda la cuadrícula le sumamos uno al número de soluciones, que hemos pasado como puntero para poder cambiar su valor.

Al finalizar este proceso, *execute_backtracking* devuelve el número de soluciones del sudoku.

Puesto que, para calcular el número de soluciones alcanza cada una de ellas, nos permite obtenerlas todas. Pudiendo, así, escribirlas en ficheros o imprimirlas por pantalla.

3.3. Generador aleatorio

En segundo lugar, nuestro programa genera aleatoriamente un sudoku y calcula el número de soluciones que tiene. Está diseñado para generar sudokus hasta encontrar uno bien planteado, es decir, que solo tenga una solución. Este será el que nos imprima por pantalla o escriba en un fichero.

Comenzamos definiendo una semilla que nos servirá para que la función *srand* la cambie y no generemos la misma secuencia de números aleatorios. Seguidamente, generamos, de manera aleatoria, el número de pistas que tendrá el sudoku, oscilando en el intervalo [17,25]. Recordamos que un sudoku debe tener un mínimo de 17 pistas para poder estar bien planteado.

Hemos creado la función *ramdon_generator*, que a partir de la semilla y el número de pistas genera los sudokus. El proceso está construido dentro de un bucle *while* que finaliza cuando conseguimos alcanzar un sudoku con una única solución, además tenemos un contador que nos informa del número de intentos que llevamos. Comenzamos generando aleatoriamente una posición para cada pista, asegurándonos de que no se repiten las posiciones y revisando con *check_in_box* que no se produzcan errores. Este proceso, a su vez, está envuelto en un bucle *while* que finaliza cuando alcanzamos el número de pistas, utilizando otro contador. Una vez puestas todos los datos iniciales, ejecutamos el procedimiento de *Backtracking* para determinar, si efectivamente, tiene una única solución. Si el número de soluciones es distinto de uno se repite el proceso, en caso contrario termina. Finalmente, imprimimos el mensaje de cuantos intentos llevamos y, en el caso de alcanzar el sudoku bien planteado, también lo imprimimos por pantalla con *write*.

Una vez obtenido un sudoku bien planteado, podemos obtener otro de la misma dificultad al permutar cifras, columnas, filas o regiones.

3.4. Nivel de dificultad

Por último, mediante la aplicación de los métodos de resolución utilizados por los humanos, explicados en el capítulo uno, el programa evalúa la dificultad del sudoku. La función *difficulty_calculator* trata de resolver el sudoku utilizando, en cada momento, la estrategia más sencilla posible para el humano, es decir, empieza intentando aplicar único desnudo, si no es posible pasaría a único oculto y así continuaría el mismo orden de los métodos hasta llegar a trío oculto. Cada vez que apliquemos un método y avancemos hacia la resolución volvemos a la primera estrategia, excepto cuando no podamos seguir avanzando con estas, emplearíamos el *Backtracking* hasta llegar a la solución final.

3.4.1. Preproceso

Antes de explicar como hemos implementado cada método y el valor que le hemos asignado como nivel de dificultad, tenemos que tener en cuenta que necesitamos saber cuales son los candidatos posibles de cada casilla en blanco. Para obtener esta información utilizamos la función *possible*, recorre la matriz *data* mediante un doble bucle *for*, parando en las celdas vacías y, con un tercer bucle, probamos cada cifra asignándole el valor a la casilla . Comprobamos con *check_box* si es un candidato a ocupar la celda y lo almacenamos en la matriz *candidates*, posteriormente restablecemos a cero el valor de la celda. Además, hemos elaborado una función, *account_fixed_boxed*, que cuenta el número de pistas iniciales. Simplemente recorre la matriz *datos* y mediante un contador iniciado en 0, va sumando 1 cuando encuentra una casilla rellena. El valor que retorna esta función será el valor inicial del contador que utilizaremos para detener el bucle *while* que envuelve los métodos, es decir, cuando las casillas rellenas sean las 81.

3.4.2. Único desnudo

Comenzamos los métodos por orden de dificultad, desde el más sencillo al más costoso. La función *naked_only*, de tipo *bool*, aplica la estrategia del único desnudo, recorriendo la matriz *data* y parando en las casillas vacías. Una vez encontrada una celda en blanco, mediante un bucle y un contador inicializado en 0, revisa la matriz de *candidates*, contando cuantos candidatos hay en esa celda y asignándole a una variable que hemos llamado *valor* la cifra que es candidata. Si el contador está en 1, mediante la función *fixed_boxed*, a la que le pasamos la fila, columna y la cifra asignada en *valor*, fijamos el número en la celda y lo borramos como candidato de ella, también eliminamos esa cifra como candidato de las celdas de la misma fila, columna y región. En otro caso, si el contador no es exactamente 1, pasamos a la siguiente celda vacía. Tras colocar un núnmero, *naked_only* devuelve verdadero saliendo así de la función y aumenta en 1 las casillas rellenas.

3.4.3. Único oculto

Cuando *naked_only* no encuentra ninguna casilla con un único candidato, saltamos a *hidden_only* que lleva a cabo la estrategia del único oulto. Este proceso hace una búsqueda del único oculto por los diferentes grupos, explicaremos como lo hemos hecho por filas, puesto que, los demás fueron implementados de forma análoga.

Teniendo en cuenta que estamos en el caso de búsqueda por fila, necesitamos definir una variable donde almacenaremos el valor de la columna en el caso de encontrar un único oculto. Utilizando un doble bucle *for*, donde el

primero hace referencia a la fila y el segundo a la cifra candidata, también haciendo uso de un contador incializado en cero, generamos un tercer bucle para las columnas. Es importante el orden de los bucles e inicializar el contador antes del bucle referido a las columnas, puesto que, primero fijamos la fila, luego la cifra y posteriormente pasamos por cada casilla de la fila contando cuantas veces aparece esta cifra. Así nos aseguramos que, en cada fila revisamos todas las cifras y que el contador empieza en cero para cada cifra.

Cuando pasamos por una casilla con el dato fijado continuamos directamente, mientras que cuando está vacía comprobamos si la cifra es candidata de esa celda, en tal caso almacenamos la columna y aumentamos en uno el contador. Una vez revisada la cifra en esa fila, es decir, finalizado el bucle de las columnas pero aún dentro del doble bucle inicial, comprobamos si el contador es exactamente 1. Esto indicaría que hay un único oculto, aplicando *fixed_boxed* fijamos el valor de la cifra en la celda que aparece, puesto que, tenemos la fila y la cifra almacenadas en las variables de los bucles y la columna almacenada en la variable definida. Además, *fixed_boxed*, como en *naked_only* borra los candidatos de la celda y elimina la cifra como candidata de las demás casillas de los diferentes grupos a los que pertenece dicha celda.

Esta función, también de tipo *bool*, devuelve verdadero en caso de fijar un valor, saliendo así de la función y restableciendo el orden de los métodos. En el caso de no encontrar un único oculto en alguna fila, buscamos por columnas y, posteriormente, por regiones, devolviendo falso en caso de no encontrar en ninguno de los grupos.

3.4.4. Intersección Línea-Región

Las siguientes estrategias no fijan cifras en las casillas,si no que, borran candidatos de la matriz *candidates*, por tanto, las hemos definido de tipo *integer* y nos devolverán el número de candidatos borrados.

El primero de los procedimientos que borra candidatos es la intersección Línea-Región. Lo dividimos en la intersección de una fila con una región y en la intersección de una columna con una región. Ambos como datos de entrada reciben la posición de la casilla inicial de la región y el número que observaremos. En el caso de la intersección fila-región, se define un vector de 1×3 que llamaremos *fila*, donde almacenamos el número de veces que aparece la cifra en la intersección de la fila con la región, por ejemplo, si nos encontramos que la cifra buscada aparece en dos casillas de la primera fila de la región *fila[0]* $= 2$. Una vez contadas las veces que aparece, nos aseguramos que se cumplen las condiciones de la estrategia, es decir, que hay una única fila de la región donde se encuentra la cifra y en más de una casilla. Esto lo conseguimos con una serie de *if*, que, dado el caso de que se cumpla, almacenamos el valor de la fila en otra variable. Este valor es el que devolverá *intersection_row_region* y, a partir

de él, *intersection_line_region* se encargará de borrar la cifra de las casillas de dicha fila que no pertenezcan a la región. De forma análoga *intersection_col_region* devuelve el valor de la columna si se dan las condiciones.

3.4.5. Conjuntos desnudos

Ahora tratamos las estrategias que trabajan con más de un candidato. Tienen una estructura similar y aplican una función en común, *cell_possibilities_acco unt*, que hace un recuento, en la matriz *candidates*, de los posibles valores que hay en cada celda vacía. La estructura consiste en pasarle dos o tres números, dependiendo de la estrategia, a las funciones auxiliares que se encargan de aplicar el método en los grupos.

Cuando *intersection_line_region* no borra ningún candidato, empleamos *naked_pair*, que recurre a sus funciones auxiliares que realizan la búsqueda del par desnudo por fila, columna y región. Primero, construimos *check_hidden_pair_box*, nos confirma que el par de números son los dos únicos candidatos de la casilla, aplicando *cell_possibilities_account* para confirmar que hay dos candidatos, en otro caso no tendría sentido continuar con la búsqueda. Las funciones auxiliares, a parte del par de cifras, reciben como datos de entrada la fila, columna o la casilla inicial de la región en el caso de la búsqueda en la región. Partiendo de esto, revisan cada casilla del grupo. Si sucede que uno de los dos números está ya fijado en una de las casilla, entonces, pasaríamos al siguiente par de números. Cuando encuentra un par desnudo, almacenamos la fila, columna o ambas, depende de la función auxiliar en la que nos encontremos, en una de las dos variables que definimos previamente, puesto que, en caso de haber un par desnudo en el grupo solo lo encontraremos en dos casillas. Encontrado el par desnudo, pasamos a eliminar esas dos cifras en la matriz *candidates* de las demás celdas del grupo.

A diferencia de *naked_pair*, para *naked_triple* debemos tener en cuenta todas las posibilidades, puesto que, no tienen porque estar los tres números en las tres celdas. Esto lo tuvimos presente al implementar *check_naked_triple_box*, contando cuantos de los tres números son candidatos en esa celda y comparamos con el valor obtenido de *cell_pissibilities_account*, si son diferentes retornamos 0, si tienen el mismo valor devolvemos 0 en el caso de que ese valor sea 1, en otro caso devolvemos la cantidad contada. Por otra parte, las funciones auxiliares trabajan de la misma forma que en par desnudo, con tres cifras en vez de dos.

3.4.6. Conjuntos ocultos

Por último, al tratar con conjuntos ocultos, en *check_hidden_pair_box* y *check_hidden_triples_box* nos limitamos a contar cuantas de las cifras, dadas como datos de entrada, aparecen en la celda. Esto hace que, a diferencia de las

funciones auxiliares de los conjuntos desnudos, debamos añadir más restricciones antes de pasar a almacenar la fila, columna o ambas en el caso de las regiones. Igual que en los métodos anteriores, si alguna de las cifras esta fijada en la casilla retornamos 0 y pasamos, directamente, a otro par o trío de cifras. Seguidamente, si hemos contado una única cifra en la celda también cambiamos de cifras. En el caso de no encontrar ninguna pasamos a la siguiente celda del grupo. Cuando la función auxiliar continua avanzando quiere decir que ha encontrado dos o tres cifras y se dispone a almacenar la posición. Puede darse el caso de que las cifras se encuentren en más casillas de las debidas, para evitar errores colocamos un contador que aumenta al almacenar la posición. Al entrar mas de dos veces en el caso del par o tres en el caso del trío, la función auxiliar retornará 0 y cambiaremos de cifras. Llegado el momento, hemos encontrado un par o trío oculto en un grupo, eliminamos en las celdas de la matriz *candidates* los números que no correspondan al par o trío.

3.4.7. Cálculo de dificultad

Finalmente, el nivel de dificultad que tendrá el sudoku viene dado por una escala del 0 al 6, donde resolverlo con *naked_only* le otorgara al sudoku el nivel 0. Si utiliza *hidden_only* alcanzará el nivel 0,5. El sudoku será de nivel 1 si es necesario aplicar *intersection_line_region* y de 1,5 si la aplica mas de una vez. En cambio, obtendrá nivel 2 cuando lleguemos a *naked_pair* y 2,5 si es necesario repetir esta estrategia. Cuando aplicamos *naked_triple* le asignamos nivel 3 y 3,5 al repetirla. Un sudoku será de nivel 4 al ser necesario utilizar *hidden_pair* y 4,5 al utilizarlo más de una vez. Asimismo, si *hidden_triple* es empleado el sudoku tendrá nivel 5 y 5,5 si se emplea más veces. El máximo nivel, 6, implicará que ha sido necesario el *backtracking* para resolverlo.

A

Apéndice

A.1. Sudoku.h

```
#ifndef __SUDOKU_H
#define __SUDOKU_H
// Sudoku.h
/************************************************************/
            Encabezados de la clase Sudoku


Esta clase además de almacenar la memoria para almacenar un
Sudoku debe incorparar la funcionalidad para operar con él
como puede ser modificar rellenar una casilla, comprobar
que es válido, leer desde fichero, etc.


/************************************************************/




class Sudoku
{
public:
static constexpr int BASE = 3;
static constexpr int N = BASE * BASE;


/// <summary>
/// Lee un fichero de entrada, el programa se para si hay un
    error
```

```
/// </summary>
void read_inputfile(const char *filename);


/// <summary>
/// Comprueba que no hay ningún número repetido en una fila,
    columna o región
/// </summary>
bool check() const;

/// <summary>
/// Comprueba que el valor de la fila row y columna col es
    correcto
/// </summary>
bool check_box(int row, int col) const;

/// <summary>
/// Ejecuta el algoritmo de backtraking, calculando todas las
    posibles soluciones
/// Devuelve el número de soluciones
/// </summary>
long long unsigned execute_backtracking();
long long unsigned execute_backtracking_generator();
void reset();

/// <summary>
/// Escribe el sudoku
/// </summary>
void write(std::ostream &os) const;
//genera un sudoku aleatoriamente
void ramdon_generator(int seed, int pistas);
//mira os posibles valores de cada casilla vacia
void possible();
//calcula la dificultad
void difficulty_calculator();
//escribe el sudoku con los posibles candidatos de las casillas
    vacias
void write_table_position(std::ostream &os);
//borra candidatos de las casillas vacias


private:
int data[N][N];  // Almaceno los números del tablero
bool candidates[N][N][N];
// Métodos privados
```

```
/// Rellena una posición con los números posibles
void fill_a_position(int pos, long long unsigned &count);
void fill_a_position_generator(int pos, long long unsigned &
    count);

/// Comprueba que el valor de la fila row y columna col no se
    repite en esa fila
bool check_box_in_row(int row, int col) const;

/// Comprueba que el valor de la fila row y columna col no se
    repite en esa columna
bool check_box_in_colum(int row, int col) const;

/// Comprueba que el valor de la fila row y columna col no se
    repite en esa región
bool check_box_in_region(int row, int col) const;

/// Cuenta las casillas que ya están rellenas bien porque se han
     fijado desde el
/// principio o porque se han ido rellenado
int account_fixed_boxed();


void fixed_boxed(int i, int j, int valor);// fija casilla

// Métodos para calcular la difucultad
bool naked_only();
bool hidden_only();
int intersection_line_region();
int naked_pair();
int naked_triple();
int hidden_pair();
int hidden_triple();

// Métodos auxiliares
int intersection_row_region(int a, int b, int k);
int intersection_col_region(int a, int b, int k);

int check_naked_pair_row(int a, int num1, int num2);
int check_naked_pair_col(int a, int num1, int num2);
int check_naked_pair_reg(int a, int b, int num1, int num2);
bool check_naked_pair_box(int fila, int col, int num1, int num2)
    ;
int cell_possibilities_account(int fila, int col);
```

```
int check_hidden_pair_box(int fila, int col, int num1, int num2)
    ;
int check_hidden_pair_row(int a, int num1, int num2);
int check_hidden_pair_col(int a, int num1, int num2);
int check_hidden_pair_reg(int a, int b, int num1, int num2);

int check_naked_triple_box(int fila, int col, int num1, int num2
    , int num3);
int check_naked_triple_row(int a, int num1, int num2, int num3);
int check_naked_triple_col(int a, int num1, int num2, int num3);
int check_naked_triple_reg(int a, int b, int num1, int num2, int
     num3);

int check_hidden_triples_box(int fila, int col, int num1, int
    num2, int num3);
int check_hidden_triples_row(int a, int num1, int num2, int num3
    );
int check_hidden_triples_col(int a, int num1, int num2, int num3
    );
int check_hidden_triple_reg(int a, int b, int num1, int num2,
    int num3);

// Para visualizar rellena tabla de posibilidades para imprimir
void tablepos(int tabla[N*BASE][N*BASE]);



};




#endif  // __SUDOKU_H
```

A.2. Sudoku.cpp

```
// Sudoku.cpp
/************************************************************/
                    Encabezados de la clase Sudoku


Esta clase además de almacenar la memoria para almacenar
un Sudoku debe incorparar la funcionalidad para operar
```

```
con él como puede ser modificar y rellenar una casilla,
comprobar que es válido, leer desde fichero, etc.


/*************************************************************/


/*-----------------------------INCLUDES-----------------------------*/

#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <ctime>
#include <chrono>
#include<random>

#include "Error.h"
#include "Sudoku.h"

void Sudoku::read_inputfile(const char *filename)
{
  std::ifstream infile(filename);
  if (!infile.is_open())
    report_error("No se ha podido abrir el fichero de entrada",
        __FUNCTION__);
  std::string line;
  int i = 0;     // contador de fila
  while (std::getline(infile, line) && i < N) {
    if (line.length() < N)
      report_error("La fila i del fichero de entrada tiene menos
          de N caracteres", __FUNCTION__);
    for (int j = 0; j < N; ++j) {
      data[i][j] = static_cast<int>(line[j]) - '0';
      if (data[i][j] < 0 || data[i][j] > N)
        report_error("Caracter no válido en  el fichero de
            entrada", __FUNCTION__);
    }
    ++i;
  }
  if (i < N)
    report_error("Pocas filas en el fichero de entrada",
        __FUNCTION__);

}
```

```
bool Sudoku::check() const
{
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      if (data[i][j]) {
        if (!check_box(i, j))
          return(false);
      }
    }
  }
  return(true);
}



long long unsigned Sudoku::execute_backtracking()
{
  long long unsigned num_sol = 0;
  fill_a_position(0, num_sol);
  return(num_sol);
}

void Sudoku::fill_a_position(int pos, long long unsigned &count)
{
  // Hemos encontrado una solución
  if (pos == N * N)
  {
    ++count;
#if 0 // def _DEBUG
    //if (!(count %1000000)) {
    std::cout << "Solución " << count << " encontrada" << std::
        endl;
    write(std::cout);

    //}
#endif
    // Aquí se podría poner que escriba la solución en un
        fichero
    return;
  }
  int i = pos / N;
  int j = pos %N;
  // Se trata de unaposición fija, no se cambia
  if (data[i][j]) {
```

```
    Sudoku::fill_a_position(pos + 1, count); // Pasamos a la
        siguiente posición
    return;
  }
  // Ponemos todos los posibles números en esa casilla

  for (int k = 1; k <= N; ++k) {
    data[i][j] = k;
    if (check_box(i, j)) {  // Si el número k se puede escribir
        en esta casilla
                // seguimos escribiendo en la casilla siguiente
                // si no continúa el bucle
#ifdef _DEBUG
      // write(std::cout);
      // std::cout << pos <<" " << std::endl;
#endif
      Sudoku::fill_a_position(pos + 1, count);
    }
  }
  data[i][j] = 0;
}

void Sudoku::write(std::ostream &os) const
{
  os << "-----------\n";
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      os << data[i][j];
    }
    os << "\n";
  }
  os << "-----------" << std::endl;
}

bool Sudoku::check_box(int row, int col) const
{
  if (!check_box_in_row(row, col)) return(false);
  if (!check_box_in_colum(row, col)) return(false);
  if (!check_box_in_region(row, col)) return(false);
  return(true);
}

bool Sudoku::check_box_in_row(int row, int col) const
{
  int value = data[row][col];
  for (int j = 0; j < N; ++j) {
```

```
    if (j == col) continue;
    if (data[row][j] == value) return(false);
  }
  return(true);
}

bool Sudoku::check_box_in_colum(int row, int col) const
{
  int value = data[row][col];
  for (int i = 0; i < N; ++i) {
    if (i == row) continue;
    if (data[i][col] == value) return(false);
  }
  return(true);
}

bool Sudoku::check_box_in_region(int row, int col) const
{
  int value = data[row][col];
  int i_begin = (row / BASE) * BASE, i_end = i_begin + BASE;
  int j_begin = (col / BASE) * BASE, j_end = j_begin + BASE;
  for (int i = i_begin; i < i_end; ++i) {
    for (int j = j_begin; j < j_end; ++j) {
      if (i == row && j == col) continue;
      if (data[i][j] == value) return(false);
    }
  }
  return(true);
}

void Sudoku::ramdon_generator(int seed, int pistas) {

  long long unsigned num_sol = 0;
  unsigned intentos = 0;
  while (num_sol != 1) {
    int count = 0;
    reset();
    while (count < pistas) {

      int pos = rand() % (N*N);
      int row = pos / N;
      int col = pos %N;
      if (data[row][col])continue;
      data[row][col] = rand() %N + 1;
      if (check_box(row, col) == false) {
        data[row][col] = 0;
```

```
        continue;
      }
      count++;
    }
    intentos++;
    num_sol = execute_backtracking_generator();
    std::cout << "intento" << intentos << "soluciones" <<
        num_sol << "\n";
  }
  write(std::cout);
}

//generamos la matriz que nos indica los posibles valores de
    cada casilla
void Sudoku::possible() {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      if (data[i][j]) continue;
      for (int k = 0; k < N; k++) {
        data[i][j] = k + 1;
        candidates[i][j][k] = check_box(i, j);
      }
      data[i][j] = 0;
    }
  }
}

void Sudoku::fixed_boxed(int fila, int col, int valor) {
  data[fila][col] = valor;
  // Borramos los candidatos de la casilla
  for (int k = 0; k < N; k++) {
    candidates[fila][col][k] = false;
  }
  // Lo borramos de la fila
  for (int j = 0; j < N; j++) {
    if (j == col) continue;
    candidates[fila][j][valor - 1] = false;
  }
  // Lo borramos de la columna
  for (int i = 0; i < N; i++) {
    if (i == fila) continue;
    candidates[i][col][valor - 1] = false;
  }
  // Lo borramos de la región
  int a = fila / BASE * BASE;
  int b = col / BASE * BASE;
```

```
  for (int i = a; i < a + BASE; i++) {
    for (int j = b; j < b + BASE; j++) {
      if (i == fila || j == col) continue;
      candidates[i][j][valor - 1] = false;
    }
  }
}

int Sudoku::account_fixed_boxed()
{
  int rval = 0;
  for (int i = 0; i < N; ++i) {
    for (int j = 0; j < N; ++j) {
      if (data[i][j]) rval++;
    }
  }
  return(rval);
}

void Sudoku::difficulty_calculator()
{
  int borrados = 0;
  float dif = 0;
  int casillas_rellenas = account_fixed_boxed();
  possible();   // La primera vez hacemos modificaciones
     actualizamos la matriz de posibilidades

  while (casillas_rellenas < N*N) {

#ifdef _DEBUG
    write_table_position(std::cout);  // Vamos imprimiento cada
       modificación que ha hecho
                 // Esto lo hacemos solo para ver que las cosas
                    van bien
#endif
    if (naked_only()) { casillas_rellenas++; continue; }
    if (hidden_only()) { casillas_rellenas++;dif = 0.5 ;
       continue; }
     borrados = intersection_line_region();
    if (borrados) {
      if (dif > 1) continue;
      else if (dif = 1) dif = 1.5;
      else dif = 1;
#ifdef _DEBUG
      std::cout << "Hemos borrado " << borrados << "
         posibilidades con interlr()" << std::endl;
```

```
#endif
      continue;
    }
    borrados = naked_pair();
    if (borrados) {
      if (dif > 2) continue;
      else if (dif = 2) dif = 2.5;
      else dif = 2;
#ifdef _DEBUG
      std::cout << "Hemos borrado " << borrados << "
          posibilidades con par_desnudo()" << std::endl;
#endif
      continue;
    }
    borrados = naked_triple();
    if (borrados) {
      if (dif > 3) continue;
      else if (dif = 3) dif = 3.5;
      else dif = 3;
#ifdef _DEBUG
      std::cout << "Hemos borrado " << borrados << "
          posibilidades con trio_desnudo()" << std::endl;
#endif
      continue;
    }
    borrados = hidden_pair();
    if (borrados) {
      if (dif > 4) continue;
      else if (dif = 4) dif = 4.5;
      else dif = 4;
#ifdef _DEBUG
      std::cout << "Hemos borrado " << borrados << "
          posibilidades con par_oculto()" << std::endl;
#endif
      continue;
    }
    borrados = hidden_triple();
    if (dif > 5) continue;
    else if (dif = 5) dif = 5.5;
    else dif = 5;
    if (borrados) {
#ifdef _DEBUG
      std::cout << "Hemos borrado " << borrados << "
          posibilidades con trio_oculto()" << std::endl;
#endif
      continue;
```

```
    }
    //if (borrados) continue;
  break;
  }
#ifdef _DEBUG
  write_table_position(std::cout);  // Vamos imprimiento cada
      modificación que ha hecho
                // Esto lo hacemos solo para ver que las cosas
                    van bien
#endif

  if (casillas_rellenas < N*N) {
    // Si llega hasta el final sin encontrar nada aplicamos el
        backtracking
  }

}

//Rellenamos la casilla en caso de solo tener un valor posible
bool Sudoku::naked_only()
{
  int valor;
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      if (data[i][j]) continue;
      int cant = 0;
      for (int k = 0; k < N; k++) {
        if (candidates[i][j][k]) {
          data[i][j] = k + 1;
          cant++;
          valor = k + 1;
        }
      }
      data[i][j] = 0;
      if (cant == 1) {
        fixed_boxed(i, j, valor);
        return(true); // Vamos a hacer que cuando encuentre uno
            salga
      }          // aunque podríamos dejar que llegara al final
    }
  }
  return(false);
}

bool Sudoku::hidden_only() {
```

```
// Buscamos único oculto por filas
int col;
for (int i = 0; i < N; i++) {
  for (int k = 0; k < N; k++) {
    int cant = 0;
    for (int j = 0; j < N; j++) {
      if (data[i][j]) continue;
      if (candidates[i][j][k]) {
        cant++;
        col = j;
      }
    }
    if (cant == 1) {
      fixed_boxed(i, col, k + 1);
      return(true);
    }
  }
}
// Buscamos único oculto por columnas
int fila;
for (int j = 0; j < N; j++) {
  for (int k = 0; k < N; k++) {
    int cant = 0;
    for (int i = 0; i < N; i++) {
      if (data[i][j]) continue;
      if (candidates[i][j][k]) {
        cant++;
        fila = i;
      }
    }
    if (cant == 1) {
      fixed_boxed(fila, j, k + 1);
      return(true);
    }
  }
}
// Buscamos único oculto en región
for (int a = 0; a < N; a += BASE) {
  for (int b = 0; b < N; b += BASE) {
    for (int k = 0; k < N; k++) {
      int cant = 0;
      for (int i = a; i < a + BASE; i++) {
        for (int j = b; j < b + BASE; j++) {
          if (data[i][j]) continue;
          if (candidates[i][j][k]) {
            cant++;
```

```
                fila = i;
                col = j;
              }
            }
          }
          if (cant == 1) {
            fixed_boxed(fila, col, k + 1);
            return(true);
          }
        }
      }
    }
    return(false);
}

int Sudoku::intersection_row_region(int a, int b, int k) {
    int vf = -1;
    int fila[BASE];
    for (int i = a; i < a + BASE; i++) {
      fila[i - a] = 0;
      for (int j = b; j < b + BASE; j++) {
        if (data[i][j]) continue;
        if (candidates[i][j][k]) {
          fila[i - a]++;
        }
      }
    }
    if (fila[0] >= 2 && fila[1] == 0 && fila[2] == 0)
      vf = a + 0;
    if (fila[1] >= 2 && fila[0] == 0 && fila[2] == 0)
      vf = a + 1;
    if (fila[2] >= 2 && fila[1] == 0 && fila[0] == 0)
      vf = a + 2;
    return vf;
}

int Sudoku::intersection_col_region(int a, int b, int k) {
    int vc = -1;
    int columna[BASE];
    for (int j = b; j < b + BASE; j++) {
      columna[j - b] = 0;
      for (int i = a; i < a + BASE; i++) {
        if (data[i][j]) continue;
        if (candidates[i][j][k]) {
          columna[j - b]++;
        }
```

```
    }
  }
  if (columna[0] >= 2 && columna[1] == 0 && columna[2] == 0)
    vc = b + 0;
  if (columna[1] >= 2 && columna[0] == 0 && columna[2] == 0)
    vc = b + 1;
  if (columna[2] >= 2 && columna[1] == 0 && columna[0] == 0)
    vc = b + 2;
  return vc;
}
int Sudoku::intersection_line_region() {
  int borrados = 0;
  for (int a = 0; a < N; a += BASE) {
    for (int b = 0; b < N; b += BASE) {
      for (int k = 0; k < N; k++) {
        int i = intersection_row_region(a, b, k);
        if (i >= 0) {
          for (int j = 0; j < N; j++) {
            if (j >= b && j < b + BASE) continue;
            if (!data[i][j] && candidates[i][j][k]) {
              candidates[i][j][k] = false;
              borrados++;
            }
          }
        }
        int j = intersection_col_region(a, b, k);
        if (j >= 0) {
          for (int i = 0; i < N; i++) {
            if (i >= a && i < a + BASE) continue;
            if (!data[i][j] && candidates[i][j][k]) {
              candidates[i][j][k] = false;
              borrados++;
            }
          }
        }
      }
    }
  }
  return borrados;
}


// Buscamos los pares desnudos
int Sudoku::naked_pair() {
  int borrados = 0;
  // Buscamos pares denudos con los dígitos num1+1 y num2+1
```

```
  for (int num1 = 0; num1 < N - 1; num1++) {
    // Asumimos que num1 < num2
    for (int num2 = num1 + 1; num2 < N; num2++) {
      // por filas
      for (int a = 0; a < N; a++) {
        borrados += check_naked_pair_row(a, num1, num2);
      }
      // por columnas
      for (int a = 0; a < N; a++) {
        borrados += check_naked_pair_col(a, num1, num2);
      }
      // por regiones
      for (int a = 0; a < N; a += BASE) {
        for (int b = 0; b < N; b += BASE) {
          borrados += check_naked_pair_reg(a, b, num1, num2);
        }
      }
    }
  }
  return borrados;
}

int Sudoku::check_naked_pair_row(int a, int num1, int num2) {
  int borrados = 0;
  int col1 = -1;  // Guarda la columna donde aparece por primera
      vez el par desnudo
  int col2 = -1;  // Guarda la columna donde aparece por segunda
      vez el par desnudo
  for (int b = 0; b < N; b++) {
    // Si num1 o num2 ya están fijados saltamos
    if (data[a][b] == num1 + 1 || data[a][b] == num2 + 1) break;
    if (check_naked_pair_box(a, b, num1, num2)) {
      // Hemos encontrado una casilla unicamente con los números
          num1 + 1 y num2 + 1
      if (col1 >= 0) col2 = b;
      else col1 = b;
    }
  }
  if (col2 >= 0) {
    // Hay dos casillas donde sólo están los números num1 + 1 y
        num2 + 1
    // Buscamos para ver si aparacen alguno de estos candidatos
        en otras casillas de la fila
    for (int b = 0; b < N; b++) {
      if (data[a][b] || b == col1 || b == col2) continue;
      if (candidates[a][b][num1]) {
```

```
        borrados++;
        candidates[a][b][num1] = false;
      }
      if (candidates[a][b][num2]) {
        borrados++;
        candidates[a][b][num2] = false;
      }
    }
  }
  return borrados;
}

bool Sudoku::check_naked_pair_box(int fila, int col, int num1,
    int num2)
{
  if (data[fila][col]) return(false);
  if (!candidates[fila][col][num1]) return(false);
  if (!candidates[fila][col][num2]) return(false);
  if (cell_possibilities_account(fila, col) != 2) return(false);
  return(true);
}

int Sudoku::cell_possibilities_account(int fila, int col)
{
  int vf = 0;
  for (int k = 0; k < N; k++) {
    if (candidates[fila][col][k]) vf++;
  }
  return(vf);
}


int Sudoku::hidden_pair() {
  int borrados = 0;
  // Buscamos pares ocultos con los dígitos num1+1 y num2+1
  for (int num1 = 0; num1 < N - 1; num1++) {
    // Asumimos que num1 < num2
    for (int num2 = num1 + 1; num2 < N; num2++) {
      // por filas
      for (int a = 0; a < N; a++) {
        borrados += check_hidden_pair_row(a, num1, num2);
      }
      // por columnas
      for (int a = 0; a < N; a++) {
        borrados += check_hidden_pair_col(a, num1, num2);
      }
```

```
      // por regiones
      for (int a = 0; a < N; a += BASE) {
        for (int b = 0; b < N; b += BASE) {
          borrados += check_hidden_pair_reg(a, b, num1, num2);
        }
      }
    }
  }
  return borrados;
}
  // devuelve 0 si la celda tiene un numero fijado o ninguno de
     los numeros es candidato
  // devuelve 1 o 2 dependiendo de sison candidatos los numeros
     que entran
int Sudoku:: check_hidden_pair_box(int fila, int col, int num1,
    int num2) {
  if (data[fila][col]) return(0);
  int rval = 0;
  if (candidates[fila][col][num1]) rval++;
  if (candidates[fila][col][num2]) rval++;
  return(rval);
}
int Sudoku::check_hidden_pair_row(int a, int num1, int num2) {
  int borrados = 0;
  int reps = 0;
  int col1 = -1; // Guarda la columna donde aparfece por
     primera vez el par oculto
  int col2 = -1; // Guarda la columna donde aparece por segunda
      vez el par oculto
  for (int b = 0; b < N; b++) {
    // Si num1 o num2 ya están fijados o si los dos no son
       candidatos de la misma casilla saltamos
    if (data[a][b] == num1 + 1 || data[a][b] == num2 + 1) return
       (0);
    int num_celda = check_hidden_pair_box(a, b, num1, num2);
    if (num_celda == 1) return(0);
    if (num_celda == 0) continue;
    //si entra más de dos veces saltamos
    reps++;
    if (reps > 2)return(0);
    // Hemos encontrado una casilla con los números num1 + 1 y
       num2 + 1
    if (col1 >= 0) col2 = b;
    else col1 = b;
  }
  if (col2 >= 0) {
```

```
    // Hay dos casillas donde están los números num1 + 1 y num2
       + 1 a la vez y no aparecen en otra casilla del grupo
    // Borramos los otros candidatos de las casillas
    for (int k = 0; k < N; k++) {
      if (k == num1 || k == num2)continue;
      if (candidates[a][col1][k]) {
        borrados++;
        candidates[a][col1][k] = false;
      }
      if (candidates[a][col2][k]) {
        borrados++;
        candidates[a][col2][k] = false;
      }
    }
  }
  return borrados;
}

void Sudoku::tablepos(int tabla[N*BASE][N*BASE]) {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      if (data[i][j]) { // En el caso que la celda ya tenga un
          número fijado
                // escribimos ese número en todas las zonas de
                   la celda
        for (int k = 0; k < N; ++k) {
          tabla[i*BASE + k / BASE][j*BASE + k %BASE] = data[i][
              j];
        }
      }
      else { // En el caso de que no tenga un número fijado
           // escribimos las posibilidades si se pueden dar y si
                no un 0
        for (int k = 0; k < N; ++k) {
          if (candidates[i][j][k])
            tabla[i*BASE + k / BASE][j*BASE + k %BASE] = k + 1;
          else
            tabla[i*BASE + k / BASE][j*BASE + k %BASE] = 0;
        }
      }
    }
  }
}

void Sudoku::write_table_position(std::ostream &os)
{
```

```
  int tabla[N*BASE][N*BASE];
  tablepos(tabla);  // Nos aseguramos que se ha creado la tabla
      de posiciones
  for (int i = 0; i < N*BASE; ++i) {  // para cada fila
    if (!(i%N))    os << "$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n
        ";
    else if (!(i%BASE)) os << "$
        ----------------------------------------$\n";
    for (int j = 0; j < N*BASE; ++j) {
      if (!(j%N))    os << "$";
      else if (!(j%BASE)) os << "|";
      if (tabla[i][j]) os << tabla[i][j];
      else os << ' ';
    }
    os << "$\n";
  }
  os << "$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$\n";
}
```

Bibliografía

[1] Becerra Tomé, A., Núñes Valdés, J., & Perea Gonzáles, J. M. (2016). Juegos y Rarezas Matemáticas ¿Cuánta Matemática hay en los sudokus?. *Pensamiento Matemático, 6 (1)*, 113-136.

[2] Crook, J.F. (2009) A Pencil-and-Paper Algorithm for Solving Sudoku Puzzles *Notices of the AMS, 56 (4)*, 460-468.

[3] Delahaye, J.P. (2006). The Science behind SUDOKU. *Scientific American*, 80-87.

[4] Felgenhauer, B., & Jarvis, F. (2005). *Enumerating possible Sudoku grids* Recuperado de: http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf

[5] Franco, J. F., Gómez Carmona, O., & Gallego R. A. (2007). Aplicación de técnicas de optimización combinatorial a la solución del Sudoku. *Scientia et technica*, (37), 151-156.

[6] McGuire, G., Tugemann, B., & Civario, G. (2013). *There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration.* Recuperado de: https://arxiv.org/pdf/1201.0749.pdf

[7] Sevkli, A.Z., & Hamza, K.A. (2018). *Soft Comput.* Recuperado de: https://doi.org/10.1007/s00500-018-3307-6

[8] Zambon, G. (2015). *Sudoku Programming with C*. Apress.

Sudoku, mathematical aspects

Sección de Matemáticas
Universidad de La Laguna

Raúl Saavedra hernández
Facultad de Ciencias · Sección de Matemáticas
Universidad de La Laguna
alu0100896800@ull.edu.es

Abstract

The mathematical aspects of Sudoku are studied in this project. First of all, we introduce the rules of the game and we observe its evolution throughout history, from its origin to the present. Then we study different resolution methods, among which we stand out the textit Backtracking and the methods of resolution by humans. Finally, we elaborate an algorithm, applying these methods, that solves and calculates all possible solutions, generates sudokus randomly and evaluates the difficulty.

1. Introduction to sudoku

Sudoku is a game that consists of filling a board of $N \times N$ with N symbols without repeating any of them in a row, column or sub-board. The most common ia a board of 9×9 with sub-boards of 3×3. Initially, some numbers are fixed in some of the boxes, called tracks.

			7					
5		1		4		7		
						9	2	8
		5			4			
						8		1
7	2					5		
6	9			8				7
								6
			9	1				

A sudoku of order 9, with the board without tracks, has 6,670,903,752,021,072,936,960 possible solutions [1].
The Origin of sudoku, possibly, comes from the square of Euler. But there is a remarcable difference, in the square of Euler you are allow to repeat in the sub-boards.
Some of the human resolution methods are:

- Naked only
- Hidden only
- Intersecction line-region
- Naked pair
- Naked triple
- Hidden pair
- Hidden triple

2. Resolution methods

We explain different resolution methods within the computational field.
Backtracking is an algorithm based on trial and error. In the first empty box, we put the first possible number and we advance. If it is not possible to continue advancing we go back and change.

Sudoku can be thought of as a linear programming problem.
Description:

- $I = \{1, ..., 9\}$ set of rows.
- $J = \{1, ..., 9\}$ set of columns.
- $K = \{1, ..., 9\}$ set of possible numbers that we can place in each box.
- $H = \{1, 4, 7\}$ values of the rows and columns in the initial boxes of a region.
- S is a set of boxes that are filled with tracks whose elements are ordered pairs (i, j).

Decision variables:

- $x_{ijk} = \begin{cases} 1, \text{ if } k \text{ is placed in the box } (i,j) \\ 0, \text{ if } k \text{ is not placed in the box } (i,j) \end{cases}$

Model:
Subject to:

$$\begin{aligned}
\sum_{k \in K} x_{ijk} &= 1 && \forall i \in I, \forall j \in J \\
\sum_{i \in I} x_{ijk} &= 1 && \forall j \in J, \forall k \in K \\
\sum_{j \in J} x_{ijk} &= 1 && \forall i \in I, \forall k \in K \\
\sum_{i=h_0}^{h_0+2} \sum_{j=h_1}^{h_1+2} x_{ijk} &= 1 && \forall h_0, h_1 \in H, \forall k \in K \\
x_{ijk} &\in \{0,1\} && \forall i \in I, \forall j \in J, \forall k \in K \\
x_{ijk} &= 1 && \forall (i,j) \in S,\ k \text{ number of the box}
\end{aligned}$$

We implemented this mathematical model in Gusek, a computer program that facilitates an open source environment where different linear programming problems can be programmed and solved. We also talk about other less effective methods like taboo search, genetic algorithms and graph coloring [2].

3. algorithm

We have developed a tool that solves sudokus by applying Backtracking and human resolution methods. It also generates sudokus randomly, trying to get them to have a unique solution and, finally, evaluates the difficulty according to how much dedication a human would used.
We chose *C++* as the programming language and worked with *Microsoft Visual Studio* as the programming environment.

References

[1] Felgenhauer, B., & Jarvis, F. (2005). *Enumerating possible Sudoku grids* Recuperado de: http://www.afjarvis.staff.shef.ac.uk/sudoku/sudoku.pdf

[2] Franco, J. F., Gómez Carmona, O., & Gallego R. A. (2007). Aplicación de técnicas de optimización combinatorial a la solución del Sudoku. *Scientia et technica*, (37), 151-156.