Sistemas Informáticos
Curso 2005-2006

Atracción Numérica Fatal
(SuDoku JADE)

Diana García Ríos
Antonio Herranz Bandrés
Juan Sanz Ruiz

Dirigido por:
Prof. Eva Ullán Hernández
Dpto. Sistemas Informáticos y Programación

Facultad de Informática
Universidad Complutense de Madrid

Cesión de derechos

Autorizamos a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Diana García Ríos Antonio Herranz Bandrés Juan Sanz Ruiz

Resumen

SuDoku es un rompecabezas que se popularizó en Japón en 1986 y se dio a conocer en el ámbito internacional en 2005. Cada puzzle es un tablero de $n^2 \times n^2$, dividido en $n \times n$ bloques (cuadrículas de $n \times n$). Un cierto número inicial de casillas contienen números del 1 al n^2. El objetivo es completar el puzzle rellenando las casillas vacías de manera que cada fila, cada columna y cada bloque contengan todos los números del 1 al n^2.

El tipo de SuDoku más conocido es el de 9×9 con las siguientes restricciones: el número de casillas inicialmente rellenas no es mayor que 30 y están situadas de forma simétrica. Los puzzles deben tener exactamente una solución y han de poder resolverse sin recurrir a métodos de prueba y error.

El objetivo principal de este proyecto es el desarrollo de un sistema de enseñanza que vaya introduciendo progresivamente al usuario en las técnicas o razonamientos lógicos necesarios para resolver los puzzles. Para ello, será necesario estudiar las estrategias de resolución, cuya complejidad determina el nivel de dificultad de un puzzle, el cual, sorprendentemente, no tiene nada que ver con el número de casillas inicialmente rellenas.

El sistema estará dotado de distintos modos de uso ofreciendo, al margen de la faceta didáctica, otro tipo de funcionalidad para que resulte de interés a aficionados de cualquier nivel. Para ello, interesa también estudiar técnicas de generación automática de tableros, para crear tableros válidos y que respondan al nivel de dificultad deseado.

Palabras clave

Sudoku, Generador, Resolutor, Patrón de Resolución, Juego, Tutor, Estrategia, *Dancing Links*, Inteligencia Artificial, C#.

Abstract

SuDoku is a puzzle that became popular in Japan in 1986 and started to be known in the international scope in 2005. Each puzzle is an $n^2 \times n^2$ board divided into boxes of $n \times n$ (grids of $n \times n$). A certain initial number of squares are filled with numbers from 1 to n^2. The goal is to complete the puzzle by filling all the empty squares in such a way that each row, column and block contains all the numbers from 1 to n^2.

The best known SuDoku type is the 9×9 SuDoku with the following constraints: the number of initially filled squares must be lower than 30 and they must be located symmetrically. Puzzles must have exactly one solution and it has to be possible solving them without using trial and error methods.

The main target of this project is the development of a teaching system which will progressively teach the user the techniques or logical reasonings used to solve puzzles. In order to achieve this goa, it will be necessary to study the solving strategies whose complexity determines the puzzle difficulty level, which has nothing to do with the initial number of filled squares.

Besides the didactic mode, the system will also offer other types of functionality, in order to be interesting for players of any level. This is the main reason to study automatic generation of puzzles: the ability of generating valid boards with the required difficulty level.

Índice general

Índice de figuras

Capítulo 1

Introducción

El juego del sudoku ha experimentado un gran auge en el último año a nivel mundial. Todos los periódicos ofrecen a sus lectores un tablero diario en su apartado de pasatiempos, y se han popularizado los libros que sólo contienen tableros de sudoku. Esto hace que vaya donde vaya uno, siempre se encuentre con alguien que está resolviendo un sudoku.

Gracias a este auge, se han desarrollado una gran cantidad de aplicaciones relacionadas con los sudokus, tanto a nivel *software* como *hardware*. Muchas empresas desarrolladoras de juegos o de software en general han aprovechado para lanzar al mercado su propio juego del sudoku, existiendo versiones para distintos sistemas operativos, incluso para teléfonos móviles.

No sólo las empresas han aprovechado este auge, sino que programadores particulares y estudiantes de informática han desarrollado su propio resolutor y generador de tableros, ya sea con fines comerciales, por ser el tema de una práctica de alguna asignatura universitaria, o simplemente por el hecho de sentir curiosidad por el juego. Todo esto ha provocado la aparición en la red de gran cantidad de foros relacionados con la programación de sudokus.

Respecto a la resolución de tableros, en el aspecto técnico existen dos corrientes: aquéllos que utilizan algoritmos de prueba y error y aquéllos que piensan que es mejor resolver tableros utilizando técnicas que simulen el razonamiento humano. De esta última nacieron los denominados *patrones de resolución*, de los cuales hablaremos en el capítulo 3.

1.1. Objetivos iniciales del proyecto

El proyecto *Atracción Numérica Fatal* se planteó en una época de explosión de popularidad del sudoku. La aparición de este juego revolucionó las páginas de pasatiempos de los periódicos de todo el mundo, y despertó la curiosidad de millones de personas.

Hoy en día, casi cualquier persona sabe en mayor o menor medida qué es un sudoku y sus reglas básicas, pero la mayoría los resuelven "por instinto",

aplicando unos razonamientos lógicos que no analizan. La curiosidad nos llevó a investigar en profundidad los razonamientos que los jugadores de sudoku más expertos seguían para su resolución.

1.1.1. Técnicas de resolución humanas

El principal objetivo perseguido en el desarrollo de nuestro sistema, SuDoku JADE, fue acercar esta metodología de razonamiento a cualquier jugador *amateur*, e introducir al usuario de manera sencilla, didáctica y gradual en las técnicas más avanzadas de resolución manual de sudokus.

Puesto que resolver un sudoku es un problema de algoritmia que cualquier programador novato resolvería con facilidad, nuestra principal preocupación fue abordar la **resolución a través de patrones**, sintetizando con ellos los razonamientos que utilizan las personas.

No obstante, entre las funciones de resolución de SuDoku JADE también interesaba incluir algoritmos de **resolución automática** de sudokus como contrapunto a la resolución por patrones y para asistir a la misma.

Otro objetivo que nos marcamos fue conseguir dar **pistas** sobre cualquier configuración del tablero, para seguirlo resolviendo y poder deducir un paso de resolución coherente en cualquier momento.

1.1.2. Versatilidad de la herramienta

Queríamos que el usuario encontrase la **herramienta sencilla, amigable y muy versátil**, de modo que SuDoku JADE constituyese la respuesta a los requerimientos más habituales que los aficionados al juego echaban de menos en otros programas.

Había en el mercado una cantidad ingente de programas de generación y resolución de sudokus, algunos muy buenos, aunque ninguno acababa de reunir las virtudes de todos ellos. SuDoku JADE trataría de reunir los aspectos positivos que iríamos encontrando en las demás herramientas, ofreciendo en un único programa el **mayor número de comodidades posibles para el aprendizaje del juego**, como el coloreado de casillas, la posibilidad de anotar candidatos en las casillas, el filtrado de candidatos, el cálculo y limpiado automático de candidatos, etcétera.

También queríamos permitir **abrir y guardar tableros** en los formatos de texto más populares en la red, para poder **importar y exportar tableros** generados por SuDoku JADE o por otros programas. Además, incluimos la posibilidad de **introducir manualmente tableros**. Toda esta versatilidad tiene como fin ayudar al usuario con cualquier tablero que haya encontrado en cualquier soporte y haya despertado su curiosidad o le haya planteado un problema.

1.1.3. Generación automática de tableros

Además, queríamos incluir la funcionalidad de generación de tableros, puesto que era evidente que algunos patrones eran caprichosos, y costaría encontrar suficientes ejemplos de tableros que los presentasen. Ya que la herramienta pretendía enseñar a través de la práctica, era deseable **crear un buen generador de tableros al que se le pudiese indicar nivel de dificultad deseado y los patrones que se querían aprender**. De ese modo resultaría muy asequible para quien usase la aplicación ir incrementando gradualmente la complicación de los sudokus a los que enfrentarse.

1.2. Definición

Sudoku es un rompecabezas matemático de colocación que se popularizó en Japón en 1986 y se dio a conocer en el ámbito internacional en 2005. El objetivo es rellenar una cuadrícula de 9×9 celdas dividida en subcuadrículas de 3×3 (también llamadas *cajas* o *regiones*) con las cifras del 1 al 9 partiendo de algunos números ya dispuestos en algunas de las celdas. No se debe repetir ninguna cifra en una misma fila, columna o subcuadrícula. Un sudoku está bien planteado si la solución es única. La resolución del problema requiere paciencia y ciertas dotes lógicas.

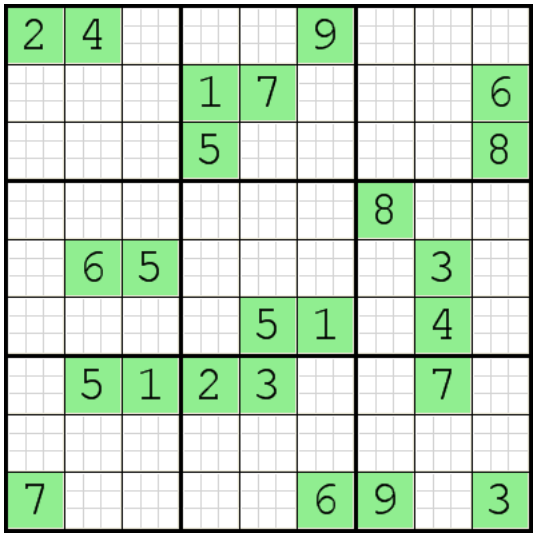

Figura 1.1: Ejemplo de tablero

En realidad, no es estrictamente necesario utilizar números, sino que se pueden utilizar letras, formas o colores sin alterar las reglas, pero se utilizan números por comodidad. La cuadrícula más común es de 9×9 con regiones de 3×3, pero también se utilizan otros tamaños. Además, las regiones no tienen por qué ser cuadradas, aunque generalmente lo son.

Numerosos periódicos empezaron a publicar sudokus en 2005 en su sección de pasatiempos.

1.3. Historia

Este rompecabezas numérico pudo haberse originado en Nueva York en 1979 cuando *Dell Magazines* publicó este puzzle, ideado por Howard Garns, bajo el nombre de *Number Place* (el lugar de los números).

Es muy probable que el Sudoku se crease a partir de los trabajos de Leonhard Euler, famoso matemático suizo del siglo XVIII. Dicho matemático no creó el juego en sí, sino que utilizó el sistema del *cuadrado latino* para realizar cálculos de probabilidades.

Cuadrado latino Es una matriz de $n \times n$ elementos, en la que cada casilla está ocupada por un símbolo de entre n posibles de tal modo que cada uno de ellos aparece exactamente una vez en cada columna y en cada fila.

La solución de un Sudoku siempre es un cuadrado latino, aunque el recíproco en general no es cierto ya que el Sudoku establece la restricción añadida de que no se puede repetir un mismo número en una región.

Posteriormente, la editorial *Nikoli* lo exportó a Japón, publicándolo en el periódico *Monthly Nikolist* en abril de 1984 bajo el título *Suji wa dokushin ni kagiru*, que se puede traducir como *los números deben estar solos*. Fue Kaji Maki, presidente de *Nikoli*, quien le puso el nombre. Posteriormente, el nombre se abrevió a Sudoku (su = número, doku = solo), ya que es práctica común en japonés tomar el primer kanji de palabras compuestas para abreviarlas.

En 1986, *Nikoli* introdujo dos innovaciones que garantizarían el éxito del rompecabezas: el número de cifras que venían dadas estaría restringida a un máximo de 30 y los puzzles serían *simétricos* (es decir, las celdas con

cifras dadas estarían dispuestas de forma rotacionalmente simétrica). Esto no siempre se cumple en los sudokus actuales. En 1997, Wayne Gould preparó algunos sudokus para el diario *The Times*, que los publicó bastante más tarde, en diciembre de 2004. Tres días después, el periódico *The Daily Mail* publicó sus sudokus con el nombre *codenumber*. En 2005 muchos otros periódicos de todo el mundo empezaron a incluir sudokus a diario en sus páginas.

En el año 2005, la ACM-ICPC *(International Collegiate Programming Contest)*, competición anual de programación y algorítmica entre universidades de todo el mundo patrocinada por IBM, incluyó entre sus 9 problemas un sudoku.

1.4. Terminología asociada

Este apartado contiene una enumeración de los términos más comunes empleados cuando se habla de sudokus, además de definiciones de interés para la comprensión de conceptos que serán tratados a lo largo de la memoria.

Casilla (también denominada celda, hueco o posición). Cuadro básico que contiene una única cifra del 1 al 9, o bien está vacía (en un juego sin resolver). Adicionalmente en estados intermedios del juego puede contener más de una cifra, como recurso para recordar las posibles cifras candidatas para rellenar esa casilla.

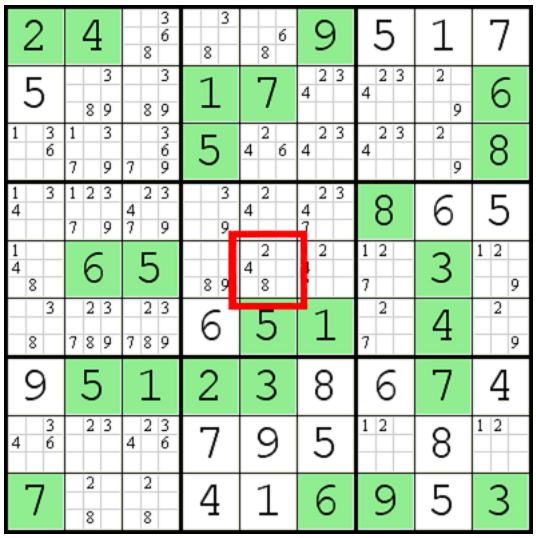

Figura 1.2: Ejemplo de casilla

Candidato (cifra candidata o candidata). Cifra del 1 al 9 que podría ocupar una celda determinada sin ser inconsistente con el estado actual de un tablero.

Tablero (también llamada cuadrícula, rejilla, rompecabezas o puzzle). Matriz de 9×9 casillas. A veces se usa la palabra sudoku para referirse a un tablero, puesto que es el recipiente de toda la información necesaria para jugar.

Bloque (utilizaremos análogamente región, caja o subcuadrícula). Matriz de 3×3 casillas, dentro de la cual no se puede repetir ninguna cifra. Desde este punto de vista, un tablero o cuadrícula es una matriz de 3×3 regiones.

Fila Secuencia horizontal de 9 celdas de un puzzle.

Columna Secuencia vertical de 9 celdas de un puzzle.

Grupo de una casilla Conjunto de las celdas que se encuentran en la misma fila, columna o bloque.

Casillas relacionadas Dos casillas c_1 y c_2 están relacionadas entre sí si se encuentran en un mismo grupo.

Par conjugado o casillas conjugadas Es aquel par de celdas que comparten un mismo grupo y son las únicas en ese grupo que contienen el candidato n. Se denominan conjugadas porque se sabe que una de las dos tendrá a n como valor final y la otra no.

Estrategia o método de resolución Pasos a seguir para conseguir completar todas las casillas de un tablero. Consta de una o varias tareas que se repiten cíclicamente y que van eliminando cifras candidatas de las casillas, de modo que llegue a quedar sólo una cifra correspondiente a cada casilla del tablero.

Patrón Disposición del tablero con unas características determinadas, reconocible independientemente del sudoku singular del que se trate, que típicamente se asociará a posibles actuaciones a llevar a cabo para alcanzar la resolución del puzzle.

1.5. Convenios

Numeración de filas y columnas

Numeraremos las filas de arriba a abajo y del 1 al 9. Adoptaremos el convenio fn para referirnos a la fila n $(1 \leq n \leq 9)$. Las columnas serán numeradas de izquierda a derecha y del 1 al 9. Adoptaremos el convenio cn para referirnos a la columna n $(1 \leq n \leq 9)$. Veáse la figura 1.5.

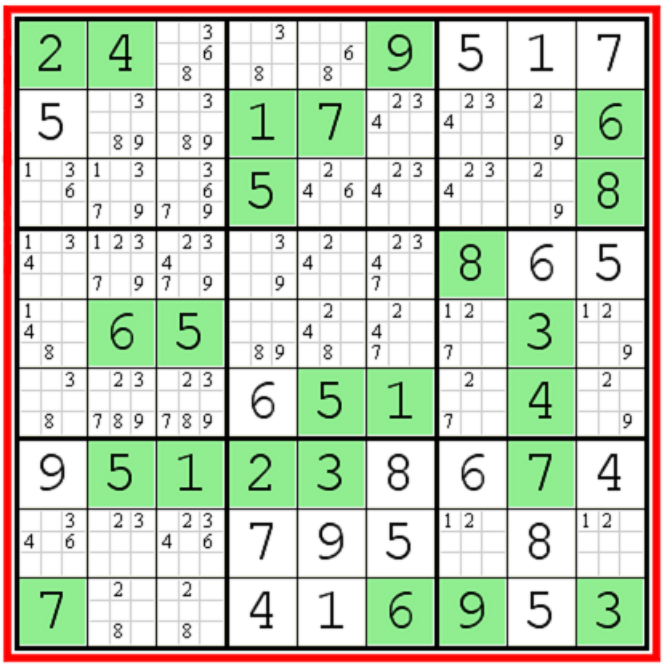

Figura 1.3: Ejemplo de tablero, puzzle o cuadrícula

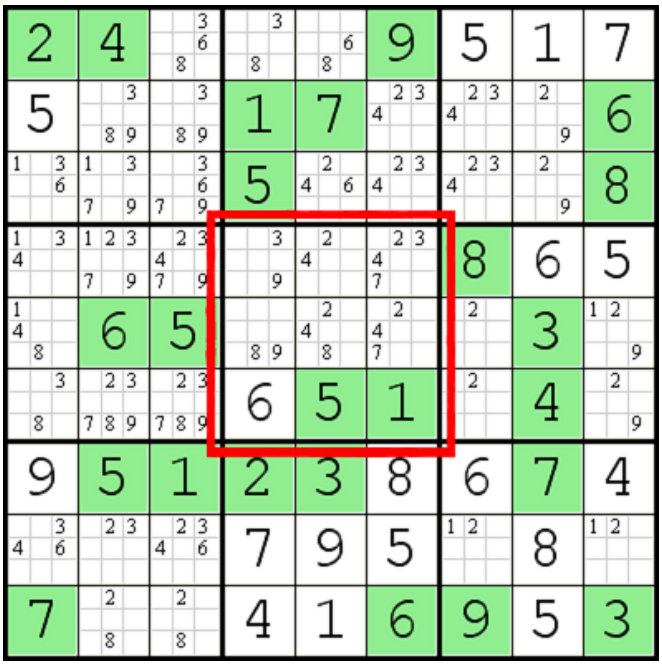

Figura 1.4: Ejemplo de bloque o subcuadrícula

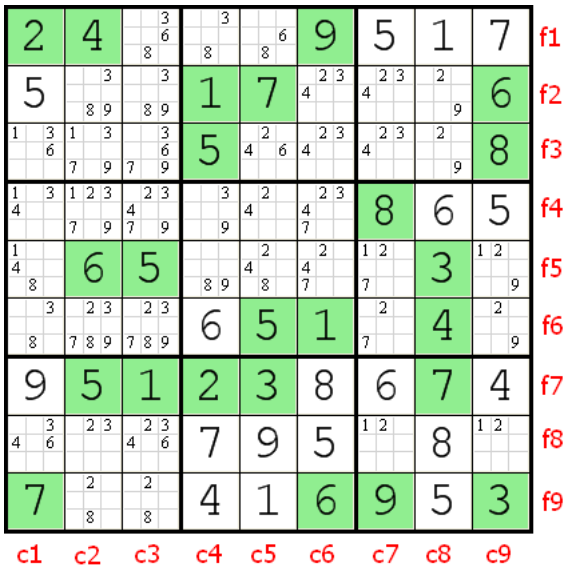

Figura 1.5: Numeración de filas y columnas

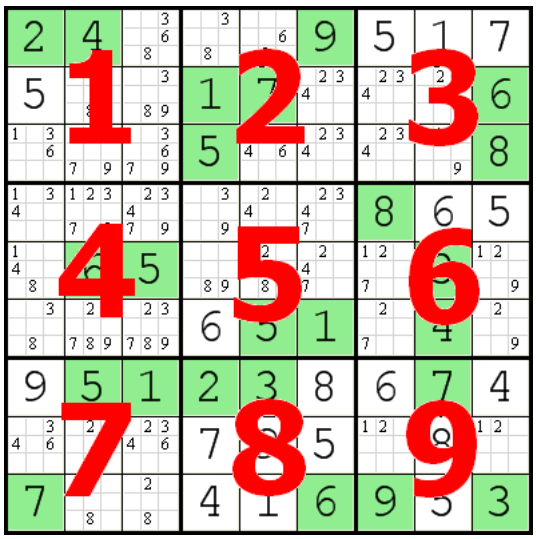

Figura 1.6: Numeración de regiones

Nomenclatura de las casillas

Basándonos en la numeración de las filas y las columnas, estableceremos un convenio para referirnos de forma breve a una casilla del tablero. Consiste en unir la numeración de la fila y la columna (por ese orden) en una palabra de cuatro letras. Por ejemplo, si nos queremos referir a la casilla que está dentro de la subcuadrícula 5 en la fila 4 y en la columna 6, utilizaremos la notación $f4c6$.

Numeración de regiones

Numeraremos las subcuadrículas del 1 al 9, tal y como se muestra en la figura 1.6.

1.6. Organización de la memoria

El siguiente capítulo describe la funcionalidad del sistema. En él encontraremos una visión general de la aplicación, seguida de una explicación detallada de todas las posibilidades que permite.

A continuación, en el capítulo 3, se presentan los patrones de resolución. Es quizá el módulo más importante de SuDoku JADE, ya que es la base que permite enseñar al usuario a resolver tableros. En él se describe cómo detectarlos y se añaden ejemplos de cada uno de ellos. Asimismo en los dos últimos patrones se explica el método seguido para su implementación, que consideramos muy interesante.

El capítulo 4 describe los distintos tipos de algoritmos de resolución mediante prueba y error implementados, destacando la aplicación a los sudokus del método *Dancing Links*.

El capítulo 5 presenta los problemas encontrados y las diversas aproximaciones que hemos seguido hasta conseguir resultados satisfactorios en la generación de tableros.

En el capítulo 6 se realiza una descripción lógica de la aplicación, así como otros detalles de implementación e instalación de la aplicación.

Finalmente, concluimos la memoria en el capítulo 7, en el que detallamos los principales problemas encontrados y cómo los hemos resuelto, enumeramos algunas ampliaciones que podrían resultar interesantes, y realizamos una comparativa con otras aplicaciones.

Capítulo 2

Funcionalidad del sistema

El principal objetivo del sistema es servir de tutor a las personas interesadas en aprender procesos deductivos que permitan resolver sudokus cada vez de mayor dificultad. La explicación de las técnicas, la creación de escenarios apropiados para su aplicación, y la demostración de cómo se detectan y aplican cumplen esta finalidad. En torno a este modo de enseñanza se han implementado diversas facilidades, así como funciones de generación y resolución de tableros.

2.1. Modos de uso

A continuación, presentamos brevemente las posibilidades que ofrece Su-Doku JADE desde diferentes puntos de vista. La forma concreta de llevarlas a cabo se explicará en apartados posteriores.

2.1.1. Uso del sistema como resolutor

Si se pretende utilizar el sistema únicamente como resolutor, haríamos lo siguiente:

1. Introducir el tablero que queremos resolver:

   a) Abriéndolo desde un archivo (en el que esté serializado en cualquiera de los diversos formatos soportados).

   b) O introduciéndolo manualmente.

   Este paso nos proporciona información acerca de si se trata de un sudoku correcto o no (no tiene o tiene más de una solución).

2. Solicitar al sistema la resolución completa del tablero, para lo que también hay dos opciones:

a) Resolverlo lógicamente por el método de los patrones, que aparte de resolverlo nos dará información de los patrones usados, y podremos saber la dificultad del sudoku.

b) O resolverlo mediante algoritmos de prueba y error si únicamente nos interesa la solución, o si el tablero es de tal dificultad que no se puede resolver lógicamente (con las técnicas conocidas hasta ahora) y requiere prueba y error.

2.1.2. Uso del sistema como generador

Si lo que se quiere es usar el sistema como generador de tableros, nuestro sistema nos proporcionará la posibilidad de **generar automáticamente un tablero del nivel de dificultad que queramos**. Una vez tengamos el tablero creado, podremos:

- Resolverlo con SuDoku JADE.
- Guardarlo en un fichero en cualquiera de los formatos más comunes de este tipo de aplicaciones.
- Imprimirlo, si queremos resolverlo sobre papel.

2.1.3. Uso del sistema como tutor

Si se va a utilizar la aplicación como tutor, un uso típico de la aplicación sería:

1. Generar un tablero de la dificultad a la que nos queremos enfrentar.
2. Resolver todos los pasos que seamos capaces. A lo largo de este proceso, podremos hacer uso de diversas facilidades como:
   - Calcular los candidatos de todas las casillas.
   - Limpiar los candidatos en conflicto con otras casillas.
   - Comprobar que los números introducidos no violan ninguna regla del sudoku.
   - Filtrar los candidatos, para que sólo muestre los que queramos, por ejemplo sólo los sietes y los nueves.
   - Aplicar colores a las casillas (para detectar patrones *Coloring* que explicaremos en el capítulo 3).
   - Deshacer o rehacer pasos.
   - Volver al estado inicial, si nos hemos equivocado y no localizamos dónde.

- Resolver todos los pasos sencillos y mecánicos (los que se resuelven con los patrones *Naked* y *Hidden Single*) para centrarnos en los que suponen más esfuerzo.

3. Si no somos capaces de seguir, podemos pedirle al sistema que nos dé una pista sobre cómo seguir, o directamente resuelva un paso explicando las deducciones lógicas que ha seguido.

Existe otro uso reseñable en la misma línea de utilización del sistema como medio de aprendizaje. Cuando se desea **aprender un patrón concreto**, lo más razonable sería hacer lo siguiente:

1. Acceder a la explicación del patrón desde el menú de ayuda.

2. Solicitar a la aplicación que genere un tablero en el que la forma más sencilla de resolverlo requiera la aplicación del patrón.

   El programa genera un tablero adecuado y lo resuelve parcialmente hasta el punto en que el patrón resulta aplicable.

3. Una vez hayamos realizado varias veces el paso anterior, deberíamos haber aprendido a detectar el patrón.

   Para comprobarlo, podemos generar tableros de la dificultad correspondiente al patrón y tratar de resolverlos.

Ahora que ya tenemos una visión general de la aplicación, iremos explicando todas las opciones que ofrece nuestro sistema.

2.2. Introducción de valores y candidatos

SuDoku JADE permite representar tanto los valores finales como los distintos candidatos que puede tener una casilla. Para representar estos últimos se divide cada casilla en nueve partes, en las que se podrán situarse los candidatos, como podemos ver en la figura 2.1.

Para insertar un candidato en una casilla se selecciona la casilla con el ratón, y se pulsa el número deseado. Para eliminarlo se sigue el mismo procedimiento, se selecciona la casilla y se pulsa el número.

Para asignar un valor a una casilla, se selecciona ésta con el ratón, y se pulsan conjuntamente la tecla "Control" y el número. Para eliminarlo se selecciona la casilla y se pulsa el número, apareciendo los candidatos que había previamente a la asignación del valor.

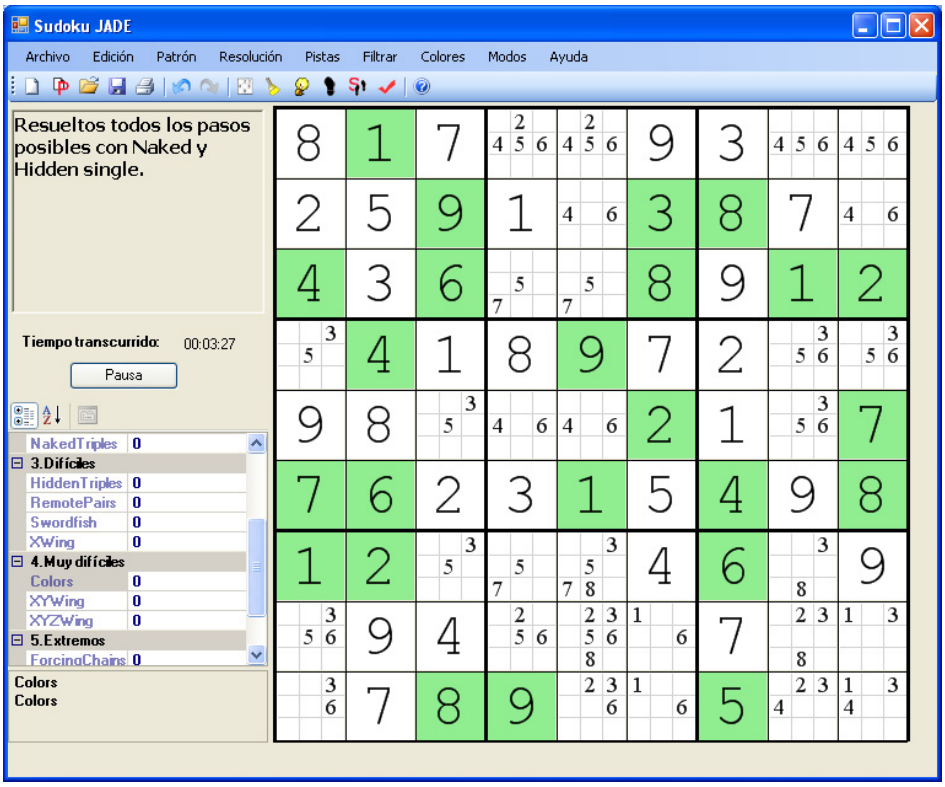

Figura 2.1: Captura de pantalla de SuDoku JADE

2.3. Los distintos menús de la aplicación

2.3.1. Menú Archivo

Generar nuevo tablero

El usuario es el encargado de elegir la dificultad del tablero a generar, la cual se mide según los patrones que se necesiten aplicar para resolverlo.

Eligiendo el nivel en el *combo* superior, se seleccionarán los patrones que puede ser necesario utilizar para resolver el tablero generado, como puede verse en la figura 2.2. Asimismo se indicará en la nota a pie de ventana el tiempo estimado en generar un sudoku de ese tipo.

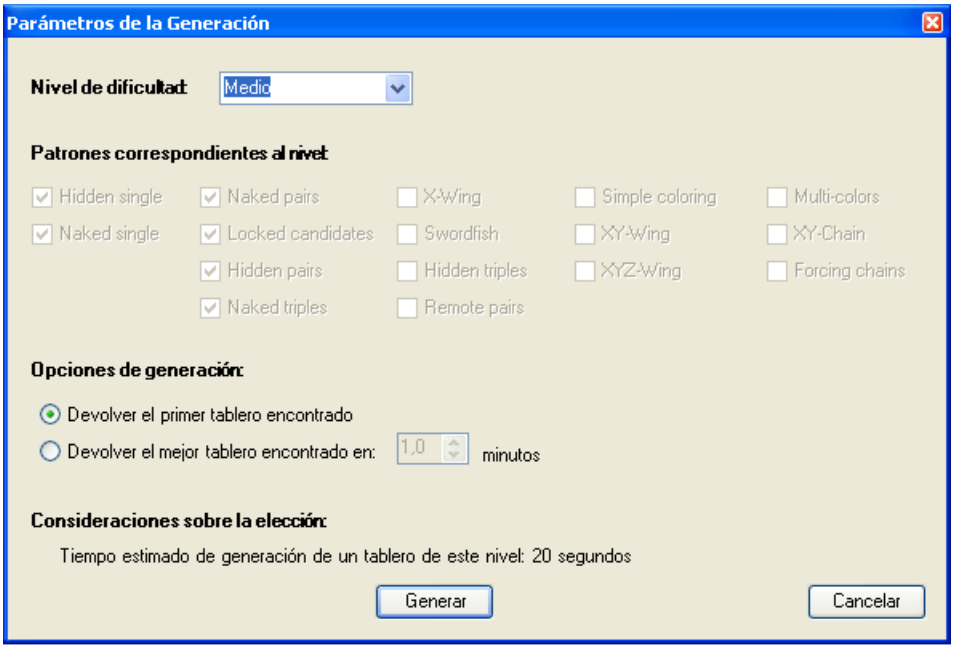

Figura 2.2: Ventana de petición de parámetros de la generación

Clasificando de este modo, nos encontramos con el siguiente problema: un tablero que necesita un paso *Naked Pairs* y el resto *Singles* será considerado de nivel medio; sin embargo un tablero que necesita cuatro *Naked Pairs*, seis *Locked Candidates*, un *Hidden Pairs* y el resto *Singles* será considerado del mismo nivel.

Teniendo esto en mente, el usuario puede elegir el grado de dificultad dentro de un mismo nivel. Por ejemplo, si quiere un tablero de nivel medio-bajo seleccionará "Medio" en el combo superior y marcará la opción "Devolver el primer tablero encontrado". Si marca la opción "Devolver el mejor tablero encontrado" el sistema dedicará el tiempo que se le indique a buscar el que tenga más apariciones de los patrones característicos del nivel medio.

Los patrones característicos de cada nivel son:

- Muy fácil:
  - Hidden Single
- Fácil:
  - Naked Single
- Medio:
  - Naked Pairs
  - Locked Candidates
  - Naked Triples
  - Hidden Pairs
- Difícil:
  - X-Wing
  - Remote Pairs
  - Swordfish
  - Hidden Triples
- Muy difícil:
  - Simple Coloring
  - XY-Wing
  - XYZ-Wing
- Extremo:
  - Multi-Colors
  - XY-Chain
  - Forcing Chains

Generar ejemplo de patrón

Esta opción genera un tablero que precise para su resolución el patrón que se seleccione. Está especialmente indicada para aprender a utilizar y a detectar el patrón elegido.

Dado que podría ser un problema presentar un tablero vacío y que hubiera pasos muy complicados antes de llegar al paso requerido, se resuelve parcialmente el sudoku para presentarlo en la situación concreta en la que se habría de aplicar el patrón.

Escribir nuevo

Permite al usuario introducir un tablero manualmente, ya sea porque lo vio en un periódico o revista de pasatiempos o por otros motivos, y quiere resolverlo usando la aplicación.

El sistema es capaz de detectar si el tablero introducido no tiene solución o, por el contrario, tiene múltiples soluciones. En estos casos se informará al usuario debidamente y se le permitirá modificarlo o bien cancelar la operación, pero en ningún momento se le dejará trabajar con el tablero.

Cuando se está en este modo, los números se introducen pulsando la tecla del número correspondiente (sin pulsar la tecla "Control"). Cambia con respecto al modo en que se introducen los valores durante el juego porque se trata de facilitar en cada caso la operación más habitual. En modo de juego, ésta es poner/quitar candidatos, mientras que en el modo de crear un tablero la operación más habitual es introducir valores.

Si en una casilla nos hemos confundido de valor, la seleccionamos y pulsamos el nuevo número. Si le hemos dado un valor cuando tenía que estar vacía, se selecciona y se pulsa la tecla "Suprimir".

Abrir modo texto

Esta operación permite abrir cualquier tablero guardado previamente por nuestra aplicación u otras (siempre y cuando esté en cualquiera de los formatos soportados por nuestra aplicación, que explicamos a continuación).

Guardar modo texto

Esta operación permite al usuario guardar el tablero actual en un archivo para poder seguir jugando o aprendiendo más adelante. Hemos dotado a nuestro sistema de la capacidad de procesar los formatos más habituales de archivos que se utilizan para guardar un tablero de sudoku.

En las descripciones que siguen, hemos añadido a modo de ejemplo cómo quedaría representado en cada formato el tablero de la figura 2.1:

.txt Es una cadena de 81 caracteres en una única fila. Aparece un punto si no hay valor definitivo, y un número del 1 al 9 en caso contrario.

En el siguiente ejemplo deberían ir todos los caracteres en una misma fila, pero la hemos cortado por falta de espacio:

```
.8.9.2..5.7.1.5.989153.862.79.5.4.8.45..
...6..2..3..5..47.9...2.39257..6.6.4...79
```

.sdk Es parecido al anterior, salvo en que se dividen los caracteres en 9 líneas de 9 caracteres cada una.

```
.8.9.2..5
.7.1.5.98
9153.862.
79.5.4.8.
45.....6.
.2..3..5.
.47.9...2
.39257..6
.6.4...79
```

.ss Este formato coincide con la representación anterior, pero se divide por bloques para que sea más fácil de leer por el usuario.

```
*-----------*
|.8.|9.2|..5|
|.7.|1.5|.98|
|915|3.8|62.|
|---+---+---|
|79.|5.4|.8.|
|45.|...|.6.|
|.2.|.3.|.5.|
|---+---+---|
|.47|.9.|..2|
|.39|257|..6|
|.6.|4..|.79|
*-----------*
```

.sud Parecido al *.sdk*, salvo que las casillas no asignadas se representan con un 0 en lugar de un punto.

```
080902005
070105098
915308620
790504080
450000060
020030050
047090002
039257006
060400079
```

.ssc Se sigue la misma filosofía que en el formato anterior, sólo que de las casillas no asignadas se guarda también la información relativa a sus candidatos.

```
*-------------------------------------------------------------*
|36     8      346    |9      467    2      |1347   134    5     |
|236    7      2346   |1      46     5      |34     9      8     |
|9      1      5      |3      47     8      |6      2      47    |
|--------------------+--------------------+--------------------|
|7      9      136    |5      126    4      |123    8      13    |
|4      5      138    |78     1278   19     |12379  6      137   |
|168    2      168    |678    3      169    |1479   5      147   |
|--------------------+--------------------+--------------------|
|158    4      7      |68     9      136    |1358   13     2     |
|18     3      9      |2      5      7      |148    14     6     |
|1258   6      128    |4      18     13     |1358   7      9     |
*-------------------------------------------------------------*
```

Imprimir tablero

Abre un asistente para imprimir el tablero con la configuración actual.

2.3.2. Menú Edición

Deshacer

Deshace el último paso realizado.

Rehacer

Rehace un paso, siempre y cuando previamente hayamos deshecho alguno.

Volver a estado inicial

Vacía todas las casillas que no formaran parte del tablero inicial. Por tanto, esta función elimina todos los movimientos que hizo el usuario, ya sean candidatos o casillas asignadas.

2.3.3. Menú Patrón

Este menú contiene todos los patrones de los que dispone la aplicación para resolver tableros. Se muestran en orden creciente de dificultad.

Si en este menú se selecciona un patrón concreto, cuando se pida al sistema generar una pista o resolver un paso, lo hará sólo utilizando este patrón. Si no se especifica ningún patrón (queda marcado "Cualquiera"), la aplicación utilizará el patrón más sencillo posible en cada paso.

2.3.4. Menú Resolución

Resolver un paso

Esta función resuelve un paso en la configuración actual del tablero. Si hay algún patrón seleccionado del menú anterior, tratará de hacerlo únicamente por ese método. Si no hay seleccionado ninguno, lo hará con el más sencillo posible.

El paso consistirá en asignar un número a una casilla o eliminar candidatos de una o varias casillas.

Además de mostrar gráficamente el paso realizado coloreando casillas o de otros modos, se incluye una breve explicación de cómo se ha deducido y del nombre del patrón utilizado. Por si se necesitara consultar la explicación del patrón, se facilita mediante un vínculo en el texto anterior un acceso directo a la parte correspondiente de la ayuda.

El resultado de ejecutar un paso sería análogo a lo que se muestra en la figura 2.3.

Resuelve singles

Aplica repetidamente la función de resolver un paso, usando los patrones *Naked Single* y *Hidden Single* hasta que el tablero queda resuelto por completo, o en su defecto, hasta que resulta necesario utilizar un patrón más avanzado.

Es muy útil si el usuario se quiere centrar en resolver los pasos complicados, ya que estos dos patrones son muy mecánicos.

Resolver completo con patrones

Resuelve el tablero completamente aplicando patrones de resolución. Su funcionamiento consiste en la utilización de la función paso, trabajando con patrones en orden de dificultad ascendente.

Una vez concluido el proceso, se informará al usuario, mediante la tabla de la esquina inferior izquierda, de cuántas veces se ha aplicado cada patrón, por lo que puede utilizarse para averiguar la dificultad de un sudoku.

Resolver completo con prueba y error

Resuelve el tablero completamente mediante un algoritmo de prueba y error. Este método garantiza que si el sudoku tiene solución, la encontrará, y es más rápido que el método anterior, pero no proporciona ninguna información sobre la dificultad del tablero.

2.3.5. Menú Pistas

Generar pista

Esta función genera una pista aplicable a la configuración actual del tablero. Si hay algún patrón seleccionado del menú anterior, tratará de encontrarla únicamente aplicando ese patrón. Si no hay seleccionado ninguno, lo hará con el más sencillo posible.

Además de mostrar gráficamente la pista, coloreando casillas o de otros modos, se incluye una breve explicación de cómo se ha deducido y el patrón utilizado. Por si se necesitara consultar la explicación del patrón, se facilita, mediante un enlace en el texto anterior, un acceso directo a la parte correspondiente de la ayuda. En la figura 2.3 puede verse un ejemplo.

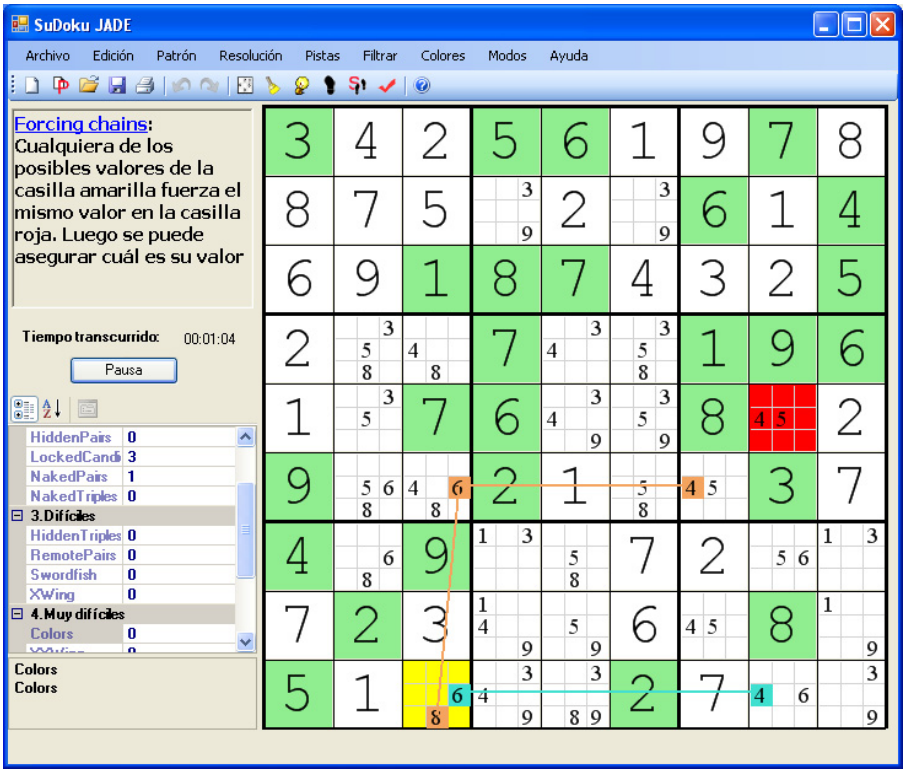

Figura 2.3: Captura de pantalla tras la generación de una pista

Limpiar candidatos sobrantes

Al aplicar esta función, se eliminan todos los candidatos sobrantes de todas las casillas.

Entendemos por candidato sobrante de la casilla c a todo número que:

- Aparece como candidato en c.

- No puede ser el valor de c, puesto que ya hay una casilla con ese número asignado en el mismo grupo.

Limpiar todos los candidatos

El resultado de esta función es la eliminación de todos los candidatos de todas las casillas que están aún sin asignar, quedando el tablero libre de candidatos.

Calcular candidatos

Calcula todos los candidatos posibles de todas las casillas que todavía no tienen un valor definitivo.

Es importante resaltar que esta función no hace uso de los patrones, por lo que si, por ejemplo, eliminamos un candidato utilizando una regla *Locked Candidates*, al calcular candidatos volverá a aparecer.

Comprobar restricciones del juego

Comprueba si todos los números asignados hasta el momento cumplen las restricciones del juego.

Si se cumplen, se informará al respecto. Si no es así, se colorearán las casillas de rojo, y seleccionando una casilla roja podrá verse qué restricción está violando.

2.3.6. Menú Filtrar

La funcionalidad de filtrar se puede aplicar a cualquier número n tal que $1 \leq n \leq 9$. Cuando se aplica el filtro a n, se ocultan de todas las casillas todos los candidatos que distintos de n.

Es posible filtrar a la vez por varios números, en cuyo caso se mostrarán sólo aquellos candidatos que formen parte del filtro, ocultando el resto.

Se puede eliminar un número del filtro de-seleccionándolo en el menú. Por ultimo, también es posible eliminar del filtro todos los números a la vez mediante la opción "Eliminar filtros".

2.3.7. Menú Colores

Menú contextual de las casillas

Dado que hemos implementado patrones que utilizan colores para representar diversas deducciones lógicas, también permitimos al usuario colorear casillas de cinco colores diferentes para facilitarle la detección de los patrones *Coloring* y *Remote Pairs*.

Figura 2.4: Menú contextual de una casilla

El coloreado de casillas se realiza a través del menú contextual que aparece al pulsar el botón derecho del ratón sobre una casilla, y que puede verse en la figura 2.4.

Seleccionando cualquiera de los colores se coloreará la casilla de dicho color. El resto de las opciones de este menú contextual se describen a continuación:

Borrar color Eliminará el color de esa casilla.

Marcar errónea Coloreará la casilla de color rojo.

Borrar todos los colores Hace lo mismo que la siguiente entrada del Menú Colores.

Borrar colores

Existen dos tipos de borrado:

Borrar todos los colores Elimina el color de todas las casillas del tablero.

Borrar color *n* Elimina el color de las casillas que tengan ese color.

Mostrar colores

Cuando esta opción está activada, las casillas coloreadas se visualizan del color que tengan asignado, mientras que si está desactivada éstas aparecerán siempre blancas.

2.3.8. Menú Modos

Elimina candidatos automáticamente

Si está activado este modo, cuando el usuario asigna un número a una casilla, automáticamente se eliminan los candidatos que entran en conflicto con el nuevo valor.

También se limpiarán los candidatos sobrantes tras cada paso resuelto por el sistema.

Modo normal

En este modo de juego el usuario es el encargado de eliminar los candidatos sobrantes siempre que se realice una nueva asignación, ya que la aplicación no lo hará por él.

2.3.9. Menú Ayuda

Mostrar ayuda

Abre el fichero *.chm* que contiene la ayuda de la aplicación. En ella se pueden encontrar las siguientes secciones:

- Introducción
- Terminología
- Manual de usuario
- Explicación de los patrones de resolución

Explicación de los patrones de resolución

Abre el fichero *.chm* directamente en la página de comienzo de la explicación de los patrones de resolución.

Acerca de SuDoku JADE

Información del sistema.

Capítulo 3

Resolución de sudokus con patrones

Se pueden definir los patrones de resolución como aquellas configuraciones de tablero a las que, por su similitud, se puede aplicar un paso de resolución idéntico. Es decir, si un grupo de casillas cumple los prerrequisitos de un patrón, entonces se podrá aplicar el paso de resolución que éste impone sobre el tablero.

El método de resolución por patrones es el más parecido al usado por las personas al resolver un sudoku, pues se basa en las mismas deducciones lógicas.

Precisamente ésta es la principal ventaja de este método respecto a otros métodos algorítmicos de fuerza bruta. Gracias a utilizar razonamientos factibles para los humanos es capaz de ayudar a resolver de forma lógica tableros ante los que una persona puede quedarse atascada, sin saber cómo seguir (o cómo empezar).

El método consiste en aplicar reiteradamente el siguiente ciclo hasta que se consigue completar el tablero:

1. Detección de un patrón aplicable a la configuración actual del tablero.
2. Adaptación del movimiento que nos sugiere el patrón al tablero actual.
3. Aplicación del movimiento (eliminar candidatos o poner un valor a una casilla).
4. Si la acción realizada fue poner un valor definitivo a una casilla, se eliminan los candidatos que están en conflicto con ella.

Los inconvenientes de la resolución por patrones consisten en que es un método más lento y costoso que los algoritmos de fuerza bruta, ya que su fin no es la eficiencia, sino buscar la pista más fácil posible para progresar en la resolución del tablero, tal y como lo haría una persona.

3.1. Patrón *Naked Single*

Es el patrón más simple de todos, el que intentan aplicar instintivamente las personas cuando tratan de resolver un sudoku por primera vez.

> Si una casilla c tiene un único candidato n (todos los demás números salvo n ya están asignados en casillas del mismo grupo), la casilla c debe tener necesariamente el valor final n.

En la Figura 3.1 podemos observar que la casilla coloreada de azul tiene como único candidato posible al número 3, ya que el resto de números se encuentran ya asignados en casillas del mismo grupo. Por tanto, se puede asegurar que 3 será su valor final.

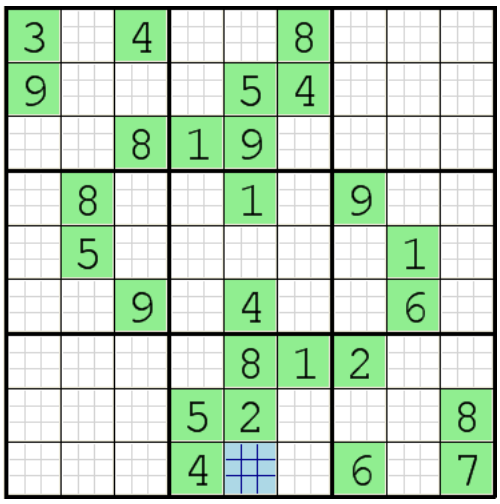

Figura 3.1: Ejemplo del patrón *Naked Single*

3.2. Patrón *Hidden Single*

El patrón *Hidden Single* se aplica a toda casilla c que tenga más de un candidato, entre ellos al número n.

> Si n no aparece como candidato en el resto de casillas de la misma fila, columna o bloque que c, entonces es lógico asignar a c el número n pues es la única casilla que puede tenerlo.

En la Figura 3.2 podemos observar que en el bloque central únicamente la casilla coloreada de azul puede tener al candidato 2, por lo que se puede asegurar que ése será su valor.

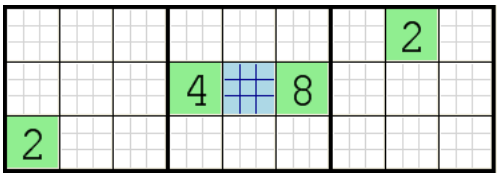

Figura 3.2: Ejemplo del patrón *Hidden Single*

3.3. Patrón *Naked Pairs*

Consideremos dos casillas c_1 y c_2, y dos candidatos n_1 y n_2. Si se cumplen las siguientes condiciones:

- c_1 y c_2 están en la misma fila, columna o bloque.
- c_1 y c_2 tienen los dos candidatos n_1 y n_2 y ninguno más.

Entonces se pueden eliminar n_1 y n_2 como candidatos del resto de casillas de la fila, columna o bloque que comparten c_1 y c_2.

El motivo es que los dos valores están restringidos a esas dos casillas. Si asignamos n_1 a c_1 tendremos que asignar n_2 a c_2, o viceversa.

En la Figura 3.3 están marcadas en marrón las casillas que forman el patrón, ya que comparten fila y las dos tienen sólo los candidatos 3 y 8. Las casillas marcadas de azul tienen como candidato al número 3 o al número 8 pero, como hemos visto, es imposible que los tengan como valor final, luego los podemos eliminar como candidatos de las casillas azules.

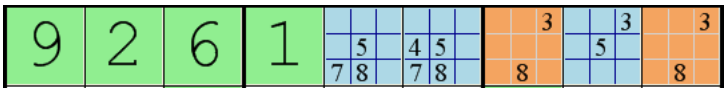

Figura 3.3: Ejemplo del patrón *Naked Pairs*

3.4. Patrón *Hidden Pairs*

Dadas dos casillas, c_1 y c_2, con más de dos candidatos, entre ellos n_1 y n_2, comunes a ambas. Si:

- c_1 y c_2 pertenecen a la misma fila, columna o bloque.
- El resto de casillas de esa fila, columna o bloque, no tienen como candidatos ni a n_1 ni a n_2.

Entonces los valores n_1 y n_2 serán valores de c_1 y c_2 y por tanto se pueden eliminar el resto de candidatos de estas dos casillas.

En la Figura 3.4 están marcadas en azul las casillas a las que se aplica el patrón. Vemos que tanto el 6 como el 7 son candidatos de ambas, y no aparecen como candidatos en el resto de casillas del bloque. Por lo tanto podemos eliminar todos los candidatos de las casillas seleccionadas, salvo el 6 y 7.

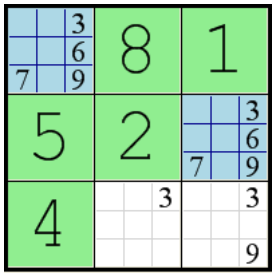

Figura 3.4: Ejemplo del patrón *Hidden Pairs*

3.5. Patrón *Locked Candidates*

Existen dos tipos de patrones *Locked Candidates*.

3.5.1. Tipo 1 o *Box Line Reduction*

Sean c_1 y c_2 dos casillas del mismo bloque b que tienen un candidato común n. Además, c_1 y c_2 pertenecen a la misma fila o columna y, en el resto de casillas de esa fila o columna no aparece n como candidato.

Entonces el número n será el valor final de c_1 o c_2 pues sólo aparece en estas dos casillas de la fila o columna a la que ambas pertenecen. Como consecuencia se puede eliminar n como candidato del resto de casillas de b sin riesgo de equivocación.

En la Figura 3.5 están marcadas en marrón las casillas que forman el patrón. Vemos que ambas pertenecen al mismo bloque y además comparten la misma columna, en la cual sólo ellas tienen al 4 como candidato. Se puede eliminar el 4 como candidato del resto de casillas del bloque, concretamente de las coloreadas de color azul.

3.5.2. Tipo 2 o *Pointing Pairs*

El razonamiento es el mismo pero a la inversa. Sean c_1 y c_2 dos casillas del mismo bloque que son las únicas del bloque con el candidato$n$. Si además pertenecen a la misma fila o columna, se puede eliminar n como candidato de todas las casillas de la fila o columna excepto de c_1 y c_2.

En la Figura 3.6 tenemos un ejemplo de este patrón. Las casillas de color marrón cumplen todas las restricciones, ya que comparten fila y están en

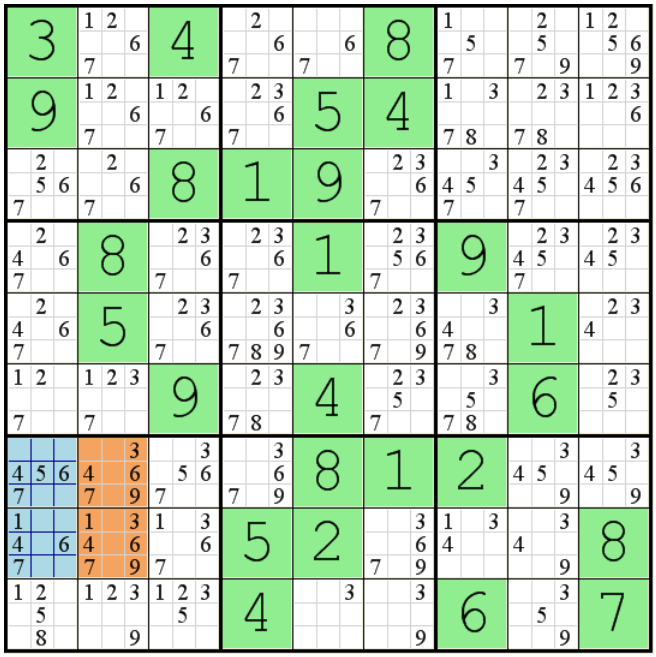

Figura 3.5: Ejemplo del patrón *Locked Candidates Tipo 1*

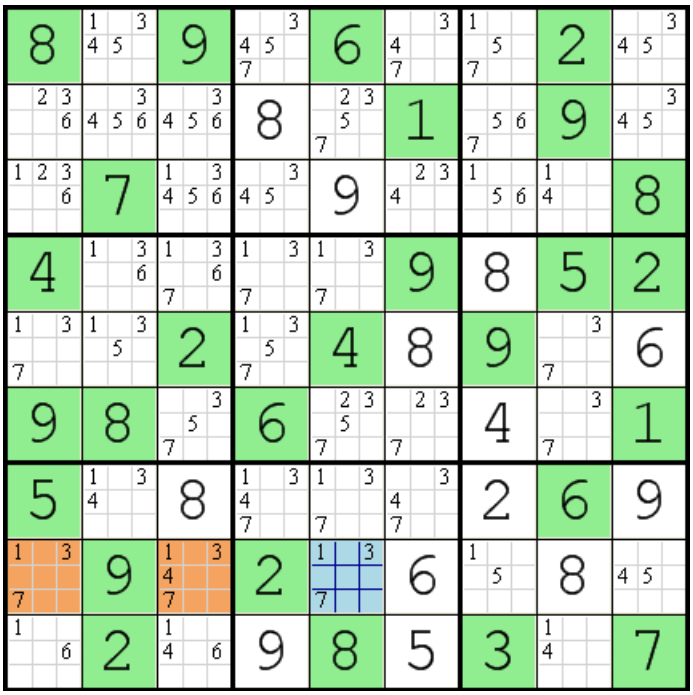

Figura 3.6: Ejemplo del patrón *Locked Candidates Tipo 2*

el mismo bloque, en el cual sólo ellas tienen el candidato 7. Por lo tanto se puede eliminar el 7 como candidato de todas las otras casillas de esa fila. Concretamente en este ejemplo se puede borrar de la casilla coloreada de azul.

3.6. Patrón *Naked Triples*

Sean tres candidatos n_1, n_2 y n_3, y tres casillas c_1, c_2 y c_3. Si se cumplen las siguientes condiciones:

- Las tres casillas se encuentran en la misma fila, columna o bloque.
- Las tres casillas tienen dos o tres candidatos, pero ninguno que no sea n_1, n_2 o n_3.

Entonces, siguiendo el mismo razonamiento que en el *Naked Pairs*, se puede asegurar que esos tres números serán asignados a esas tres casillas, y se pueden eliminar como candidatos del resto de casillas de la fila, columna o bloque que éstas comparten.

En la Figura 3.7 están marcadas en marrón las casillas que forman el patrón. Vemos que las tres pertenecen al mismo bloque, donde sólo ellas tienen a los candidatos 1, 2 y 6. Las casillas marcadas en azul tienen uno o varios de estos candidatos, de las que se podrían eliminar con toda seguridad.

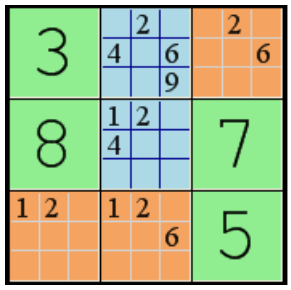

Figura 3.7: Ejemplo del patrón *Naked Triples*

3.7. Patrón *Hidden Triples*

Sean tres casillas c_1, c_2 y c_3, con dos o más candidatos. Si:

- Las tres casillas están situadas en la misma fila, columna o bloque.
- Existen tres candidatos, n_1, n_2 y n_3, que son comunes a las tres casillas (aunque no tienen por qué aparecer todos en todas).
- n_1, n_2 y n_3 no aparecen en el resto de casillas de la fila, columna o bloque que comparten las tres casillas.

Entonces, por el mismo principio descrito en el patrón *Hidden Pairs*, podemos eliminar de las casillas c_1, c_2 y c_3 todos los candidatos salvo n_1, n_2 y n_3.

En la Figura 3.8 están marcadas en azul las casillas que forman el patrón. Las tres pertenecen al mismo bloque, donde sólo ellas tienen a los candidatos 1, 4 y 6. Por lo tanto se pueden eliminar todos los candidatos de estas casillas salvo estos tres.

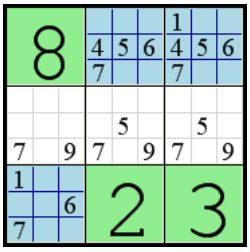

Figura 3.8: Ejemplo del patrón *Hidden Triples*

3.8. Patrón *X-Wing*

Sean dos filas f_1 y f_2, y un número n. Si se cumplen las condiciones siguientes:

- n sólo se encuentra como candidato en dos casillas de la fila f_1 y otras dos de la fila f_2.
- Esas cuatro casillas coinciden exactamente en dos columnas c_1 y c_2.

Entonces se puede eliminar el candidato n de todas las casillas de las columnas c_1 y c_2, excepto en las filas f_1 y f_2.

Este patrón recibe el nombre de *X-Wing* por formar las 4 casillas las puntas de una X. Del modo en que se ha definido encontraremos los llamados *X-Wing verticales*. Se pueden encontrar también *X-Wing horizontales*, para lo cual no hay más que cambiar en la definición anterior la palabra fila por columna, y viceversa.

En la figura 3.9 tenemos una situación real en la que se puede aplicar este patrón. Las casillas de color marrón forman un patrón *X-Wing* ya que cumplen todos los requisitos:

- Las filas 1 y 7 tienen sólo dos casillas con el candidato 2.
- Las cuatro casillas coinciden en sólo dos columnas (la 2 y la 4).

Por tanto se puede eliminar el candidato 2 de todas las casillas de las columnas 2 y 4, excepto de las casillas que forman el patrón. El resultado sería

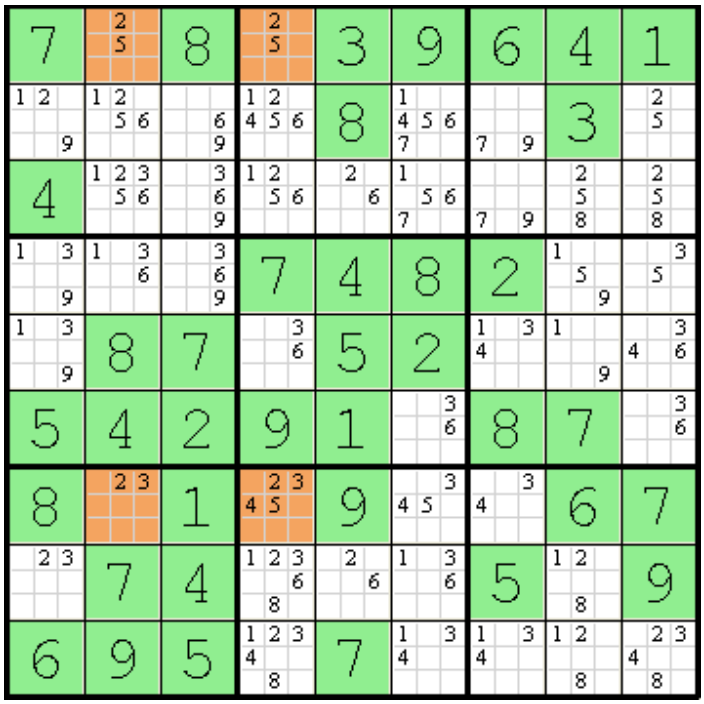

Figura 3.9: Detección del patrón *X-Wing*

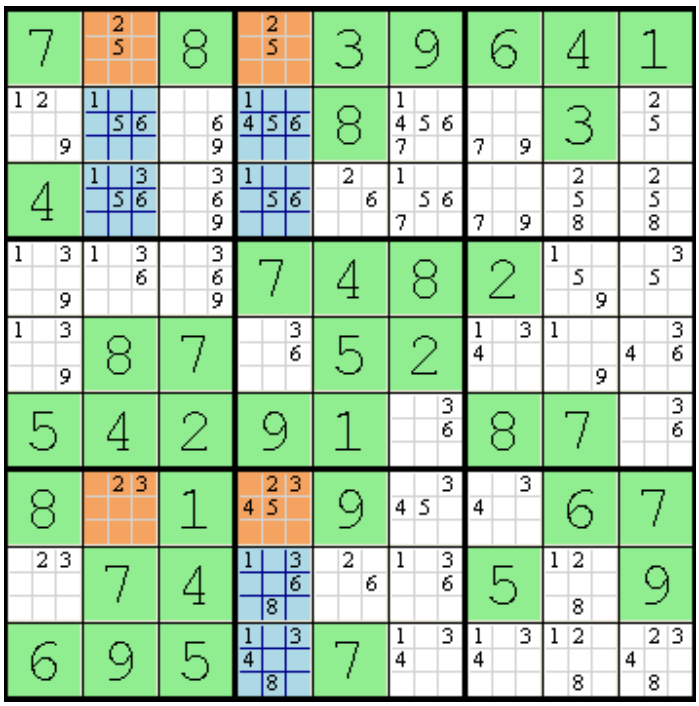

Figura 3.10: Aplicación de *X-Wing*

el de la figura 3.10, donde las casillas azules se corresponden con casillas en las que se ha eliminado el candidato.

Ahora que ya sabemos detectar el patrón, veamos los motivos que nos permiten realizar estas eliminaciones:

- Las filas 1 y 7 deben tener necesariamente un 2 cada una.
- Si en la fila 1 el valor 2 está en la casilla $f1c2$, en la fila 7 necesariamente estará en la casilla $f7c4$.
- Si en la fila 1 el valor 2 está en la casilla $f1c4$, en la fila 7 necesariamente estará en la casilla $f7c1$.
- Así tenemos dos posibilidades, una de las cuales debe ser cierta, y ambas excluyen el candidato 2 del resto de casillas de las columnas 1 y 4.

Nótese que el valor 2 será compartido por las casillas situadas en los extremos de una sola de las aspas de la X que forma este patrón. Por ello, se puede garantizar que el 2 no aparecerá en ninguna de las demás casillas de las columnas en las que están situadas los extremos de las aspas.

3.9. Patrón *Swordfish*

El patrón *Swordfish* es una generalización del patrón anterior en el que se tienen en cuenta tres filas y tres columnas.

Sean tres filas, f_1, f_2 y f_3, en las cuales todas las apariciones del candidato n coinciden bajo tres columnas. Se puede eliminar el candidato n de estas tres columnas, excepto de las casillas situadas en las filas f_1, f_2 y f_3.

Al igual que en el patrón *X-Wing*, esta definición se correspondería con un *Swordfish vertical*. Para encontrar un *Swordfish horizontal* no hay más que cambiar de la anterior definición la palabra fila por columna, y viceversa.

En la figura 3.11 podemos ver un ejemplo. Las filas 5, 8 y 9 tienen todas las casillas con el candidato 7 (coloreadas de marrón) bajo exactamente 3 columnas, luego forman un *Swordfish vertical*. Se podrá eliminar el candidato 7 de todas las casillas de esas tres columnas, excepto de las que forman el patrón.

Podemos ver el resultado de la aplicación de este patrón en la figura 3.12, donde están coloreadas en azul las casillas en las que se eliminó el candidato. El razonamiento es análogo al del patrón *X-Wing*. Las filas 5, 8 y 9 deben tener un 7 cada una, y hay sólo dos posibilidades:

- El 7 se encuentra en las casillas $f5c6$, $f8c2$ y $f9c9$.
- El 7 se encuentra en las casillas $f5c9$, $f9c2$ y $f8c6$.

Figura 3.11: Detección del patrón *Swordfish vertical*

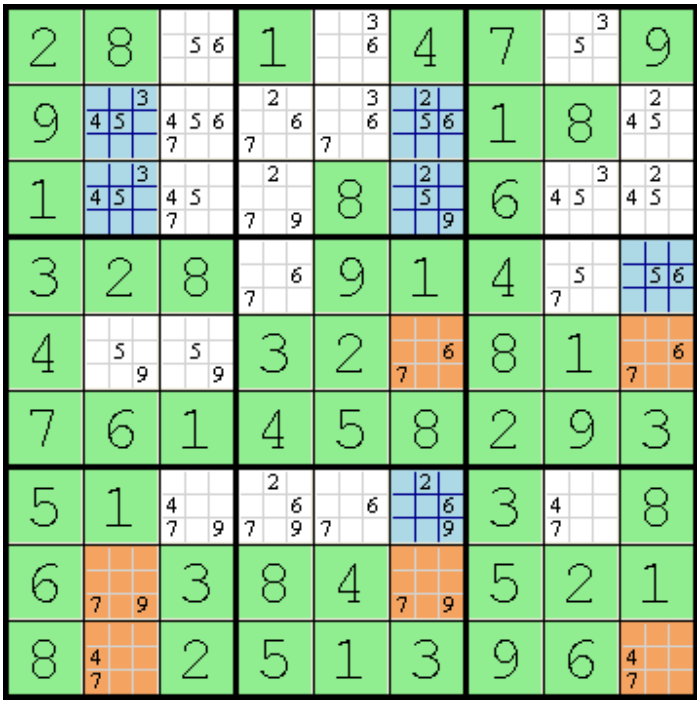

Figura 3.12: Aplicación del patrón *Swordfish vertical*

Una de las dos suposiciones es necesariamente cierta, y como ambas excluyen al candidato 7 del resto de casillas de las 3 columnas, podemos eliminarlo de éstas.

Nótese que la definición del patrón no se limita a que haya solamente dos casillas por fila. La única restricción es que coincidan todas bajo tres columnas, por lo que puede haber filas con 3 casillas.

3.10. Patrones *Coloring*

En esta sección presentamos dos patrones basados en la formación de cadenas lógicas de deducción. Estas cadenas se formarán sobre celdas que contengan un determinado candidato.

3.10.1. Definiciones

Las siguientes definiciones de conceptos hacen referencia a un candidato n:

Par conjugado o casillas conjugadas Es aquel par de celdas que comparten un mismo grupo y son las únicas en ese grupo que contienen al candidato n. Se denominan conjugadas porque se sabe que una de las dos tendrá a n como valor final y la otra no. Puede verse un ejemplo en la figura 3.13.

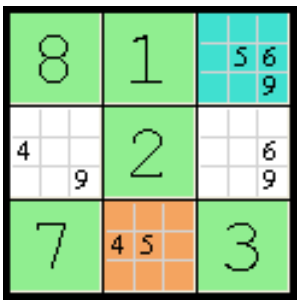

Figura 3.13: Par conjugado en el bloque para el candidato 5

Casilla cierta Casilla que tiene como valor final al candidato n.

Casilla falsa Casilla que no tiene como valor final al candidato n.

Cadena conjugada Es una cadena de pares conjugados, que se forma del siguiente modo: se parte de un par conjugado para el candidato n, y se deben añadir las celdas conjugadas (para el candidato n y sin repeticiones) de cada celda de la cadena (de las dos iniciales y de las nuevas que se añadan).

Color cierto Un color es cierto si todas las casillas coloreadas de ese color son ciertas.

Color falso Un color es falso si todas las casillas coloreadas de ese color son falsas.

Colores conjugados Dos colores distintos son conjugados si representan casillas conjugadas.

El nombre de la técnica, *Coloring*, procede de su representación gráfica, ya que lo más usual es aplicarla utilizando dos colores para representar cada cadena conjugada. Las casillas coloreadas por un mismo color serán o bien todas ciertas o bien todas falsas. Pueden verse ejemplos en la figura 3.14 y a lo largo de la explicación del patrón *Simple Coloring*.

3.10.2. Patrón *Colors* o *Simple Coloring*

Esta técnica consiste en la formación de una sola cadena conjugada para extraer conclusiones. Al haber dos tipos de deducciones posibles, para facilitar la comprensión distinguiremos dos tipos de *Simple Coloring*.

Simple Coloring Tipo 1

Sea una cadena conjugada para el candidato n. Si hay casillas coloreadas del mismo color en un mismo grupo, sabemos que ese color es falso y por tanto se pueden eliminar los candidatos n de todas las casillas de ese color.

El motivo es que sabemos que un color es o bien cierto o bien falso. Si dos casillas de un mismo color comparten grupo, no pueden ser ambas ciertas ya que violarían las reglas del juego, luego el color debe ser falso.

En la figura 3.14 podemos ver un ejemplo de este patrón, en el que se aplicaron colores al 6 para formar la cadena conjugada. Dado que hay dos casillas azules en la misma fila, sabemos que el color azul es falso, y podemos eliminar el candidato 6 de todas las celdas de ese color. Como efecto colateral de las eliminaciones, todas las casillas marrones serán ciertas (aplicando *Hidden Singles*).

Simple Coloring Tipo 2

Sea una cadena conjugada para el candidato n en la que no se cumple el *Simple Coloring Tipo 1*. Si hay alguna casilla (externa a la cadena o no) que comparte grupo con casillas de distinto color, esa casilla es falsa y se puede eliminar su candidato n.

El motivo es que uno de los dos colores conjugados es necesariamente cierto, y como la casilla se relaciona con ambos, sea cual sea el color cierto impedirá que esta casilla tenga el valor sobre el que se colorea.

En la figura 3.15 podemos ver un ejemplo de aplicación de este patrón. Se creó una cadena conjugada para el candidato 5 sin encontrar un *Simple Coloring Tipo 1*. Sin embargo hay dos casillas externas a la cadena (las

Figura 3.14: Cadena conjugada para 6 (*Simple Coloring Tipo 1*)

coloreadas en rojo) que cumplen la condición necesaria, por tanto son falsas y se podrá eliminar el candidato 5 de ambas.

3.10.3. Patrón *Multi-Colors*

Este patrón actúa también sobre casillas con el mismo candidato, pero se distingue del anterior por necesitar dos cadenas conjugadas para extraer conclusiones. Como ocurría en el patrón anterior, también hay dos tipos de aplicación bastante diferenciados.

Multi-Colors Tipo 1

Sean dos cadenas conjugadas para el candidato n. Si un color de una cadena conjugada está relacionado con los dos colores de la otra cadena conjugada, entonces ese color es falso y se pueden eliminar los candidatos n de las casillas de ese color.

Para que un color esté relacionado con los otros dos de la otra cadena no es necesario que lo haga a través de una sola casilla. Si las casillas c_1 y c_2 están coloreadas de $color_1$, este color estará relacionado con $color_3$ y $color_4$ si c_1 comparte grupo con una casilla de $color_3$ y c_2 comparte grupo con una casilla de $color_4$.

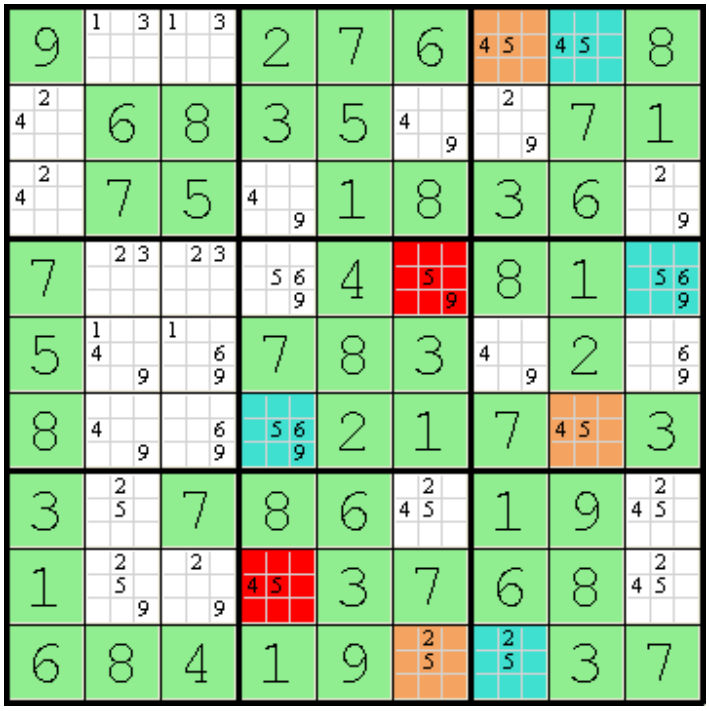

Figura 3.15: Cadena conjugada para el 5 y detección de casillas externas que cumplen el patrón *Simple Coloring Tipo 2*

En el ejemplo de la figura 3.16 se han aplicado colores al candidato 7 y se han formado dos cadenas conjugadas. Una de ellas se ha representado con los colores azul y marrón, y la otra con los colores amarillo y verde claro. El color verde claro está relacionado con los colores conjugados azul y marrón, y sabemos que uno de ellos es cierto. Luego podemos deducir que el verde claro es falso y se puede eliminar el 7 como candidato de las casillas coloreadas de este color.

Multi-Colors Tipo 2

Sean dos cadenas conjugadas para el candidato n. La primera está representada con los colores $color_1$ y $color_2$, y la segunda con $color_3$ y $color_4$. Puede afirmarse que una casilla c es falsa si se dan las siguientes condiciones:

- Alguna casilla de $color_1$ comparte grupo con alguna casilla de $color_3$.
- La casilla c comparte grupo con casillas de color $color_2$ y $color_4$

En el ejemplo de la figura 3.17 se ha detectado un patrón *Multi-colors Tipo 2*. Se han aplicado colores al candidato 6. La primera cadena conjugada se ha representado con los colores azul y marrón, y la segunda con los colores

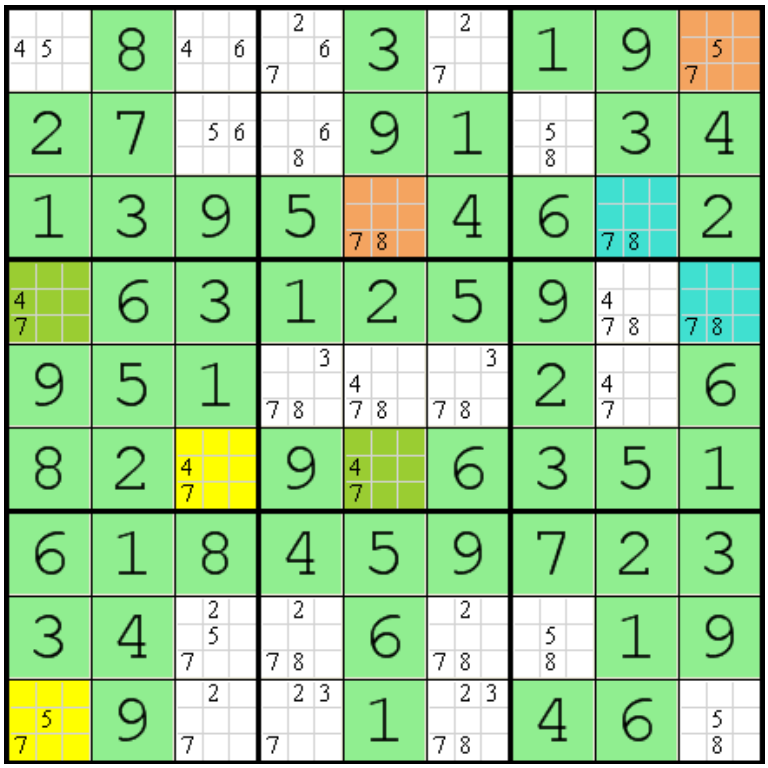

Figura 3.16: Detección de un patrón *Multi-Colors Tipo 1*

amarillo y verde claro. El color verde y el azul están relacionados y son de distintas cadenas conjugadas. Así, si hay alguna casilla que comparta grupo con los colores amarillo y marrón será falsa para el candidato 6. En el ejemplo estas casillas son las coloreadas de rojo.

Veamos por qué las casillas rojas son falsas. Sabemos que o bien el color marrón o el color azul son ciertos porque forman parte de la misma cadena conjugada. Lo mismo ocurre con el amarillo y el verde claro.

- Si el color marrón es cierto, las casillas rojas son falsas aplicando las reglas del sudoku.
- Si el color marrón es falso, el color azul debe ser cierto. Esto hace que el color verde claro sea falso, ya que comparte grupo con el azul. Por tanto el amarillo es cierto, lo que hace que las casillas rojas sean falsas aplicando las reglas básicas del sudoku.

3.11. Patrón *Remote Pairs*

Este patrón es una extensión del *Naked pairs*, utilizando las ideas de los patrones *Coloring*. No es muy frecuente, pero es bastante fácil de detectar y

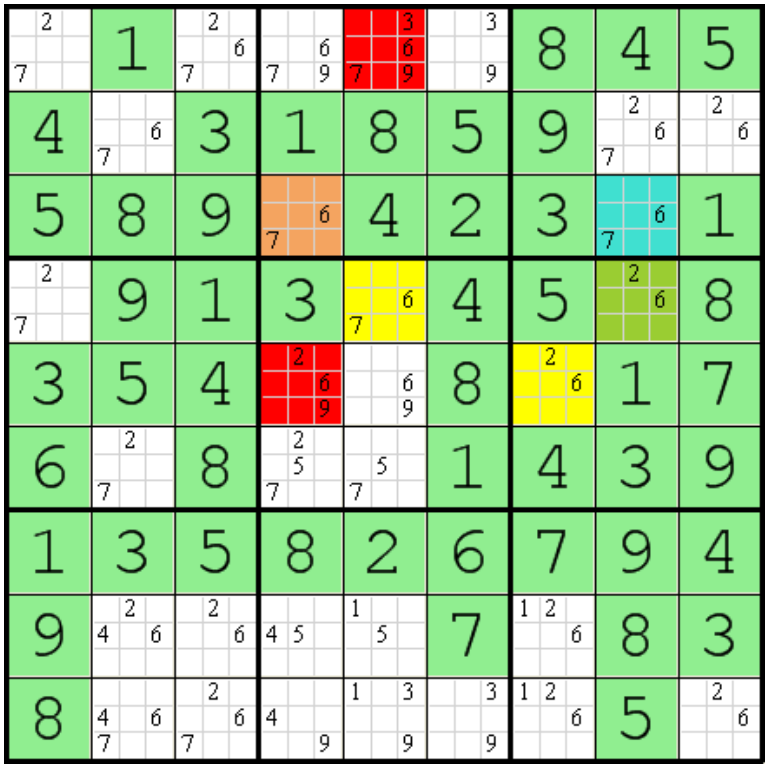

Figura 3.17: Detección de un patrón *Multi-Colors Tipo 2*

evita aplicar patrones más complicados, por lo que nos ha parecido apropiado incluirlo en el proyecto.

Consiste en aplicar colores con la idea de las casillas conjugadas, tal y como se ha visto en los patrones de *Coloring*, sólo que esta vez únicamente se colorean las casillas con los mismos dos candidatos, y sólo esos dos.

Sea una cadena conjugada sobre casillas con los candidatos n_1 y n_2. Uno de los colores representará las casillas que tendrán el valor n_1, y el otro color representará las casillas de valor n_2. Se pueden eliminar los candidatos n_1 y n_2 de cualquier casilla externa a la cadena, siempre que esté relacionada con dos casillas de distinto color.

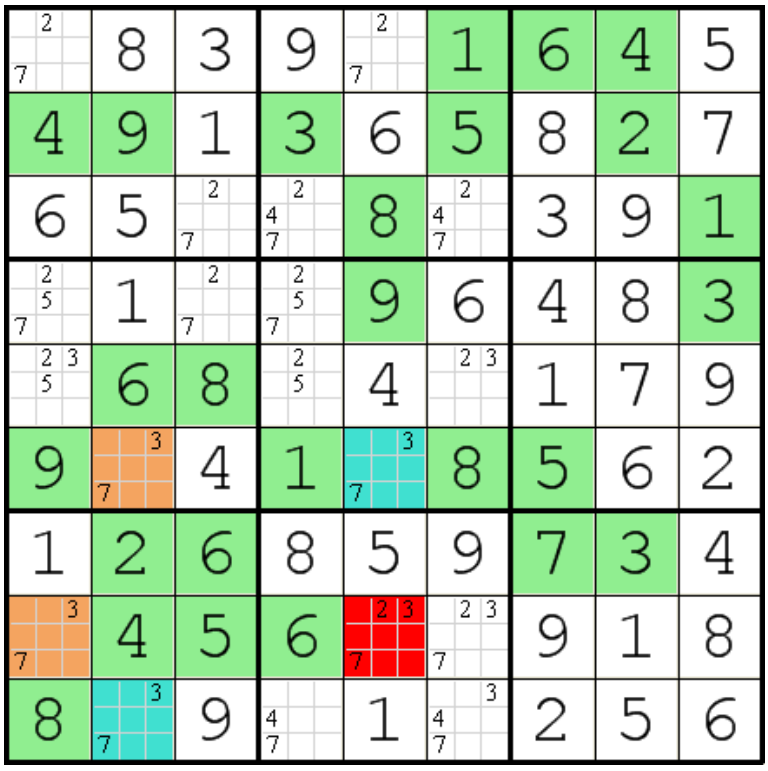

Figura 3.18: Detección de un patrón *Remote Pairs*

En la figura 3.18 podemos ver un ejemplo de este patrón. Aplica colores a las casillas con los candidatos 3 y 7. Hay dos posibilidades:

- El color azul corresponde a casillas con el valor final 3 y el color marrón a casillas con el valor final 7.
- El color azul corresponde a casillas con el valor final 7 y el color marrón a casillas con el valor final 3.

Hay una casilla (coloreada de rojo) externa a la cadena que está en el mismo grupo de casillas de ambos colores. No sabemos a qué candidato correspon-

derá cada color, pero no importa ya que cualquiera de los casos anteriores descarta los candidatos 3 y 7 para esa casilla y se pueden eliminar.

3.12. Patrón *XY-Wing*

Sean tres candidatos x, y, z. Si hay tres celdas que cumplen las siguientes condiciones:

- Tienen todas exactamente dos candidatos.
- Comparten los mismos candidatos de la forma xy, yz y xz.
- La celda con los candidatos xy (llamada *celda raíz*) comparte grupo con las otras dos celdas (llamadas *celdas ramas*).

Entonces se puede eliminar el candidato z de cualquier otra celda que comparta grupo con las dos celdas ramas.

El motivo es que si una celda que comparta grupo con las dos *celdas ramas* tuviera el valor z, entonces ninguna de ellas podría tenerlo. Por tanto, una tendría el valor x y la otra el valor y, dejando a la celda raíz sin valor posible.

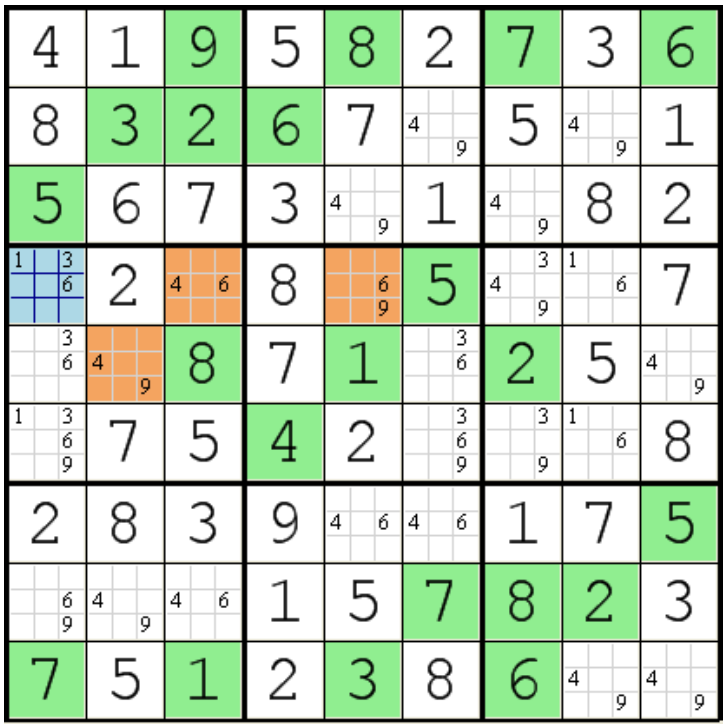

Figura 3.19: Aplicación de un *XY-Wing*

En la figura 3.19 podemos ver un ejemplo en el cual se han coloreado de marrón las casillas que forman el patrón, siendo:

- $x = 4,\ y = 6,\ z = 9$
- $xy = f4c3,\ yz = f4c5,\ xz = f5c2$

La casilla azul $f4c1$ comparte grupo con las dos celdas ramas, por lo que se ha eliminado su candidato 9.

Un caso especial de este patrón es que las tres celdas que lo forman compartan grupo. En ese caso formarían un *Naked Triple*.

3.13. Patrón *XYZ-Wing*

Es una variación del *XY-Wing* en la que una de las casillas tiene tres candidatos.

Sean tres candidatos x, y, z. Si hay tres celdas que cumplan las siguientes condiciones:

- Dos de ellas tienen exactamente dos candidatos y la otra tiene tres.
- Comparten los mismos candidatos de la forma xyz, yz y xz.
- La celda con los candidatos xyz (*celda raíz*) comparte grupo con las otras dos celdas (*celdas ramas*).

Entonces se puede eliminar el candidato z de cualquier otra celda que comparta grupo con las tres casillas que forman el patrón.

El motivo es que si alguna casilla c que comparta grupo con las tres tiene el valor z, las dos celdas ramas tendrían x e y como valor, dejando a la raíz sin valor posible, pues también comparte grupo con la casilla c.

En la figura 3.20 se han coloreado de marrón las casillas que forman el patrón, siendo:

- $x = 4,\ y = 9,\ z = 2$
- $xyz = f7c2,\ yz = f8c3,\ xz = f4c2$

La casilla azul $f9c2$ comparte grupo con las tres casillas que forman el patrón, por lo que se ha eliminado su candidato 2.

3.14. Patrón *XY-Chain*

Este patrón es un caso especial del *Forcing Chains*, que veremos en el siguiente apartado. Al tener sólo en cuenta casillas con dos candidatos, es más sencillo y además es bastante frecuente.

Hay varios métodos para detectarlo, entre los cuales hemos elegido el de los *bivalue graphs*. Al principio puede parecer algo complicado pero garantiza encontrar un patrón de este tipo siempre que lo haya.

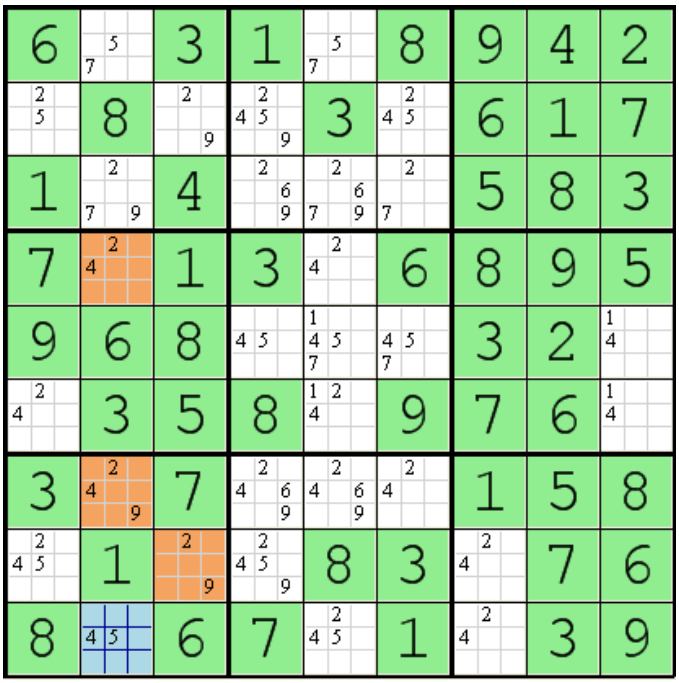

Figura 3.20: Aplicación de un *XYZ-Wing*

3.14.1. *Bivalue graphs*

Un *bivalue graph* (grafo bivalorado) es un grafo que se puede construir a partir de las casillas sin valor definitivo de un tablero de sudoku parcialmente resuelto.

A partir de un tablero que tenga casillas con exactamente dos candidatos, como el que aparece en la figura 3.21, podemos crear un grafo bivalorado aplicando el siguiente proceso:

1. Crear un vértice por cada casilla que tenga exactamente dos candidatos.

2. Conectar dos vértices con una arista etiquetada con el valor n si los dos vértices comparten grupo y ambos tienen a n como uno de sus candidatos.

En la figura 3.22 se muestra el grafo bivalorado que se obtiene a partir del anterior tablero. Las aristas que aparentemente están etiquetadas con dos dígitos son en realidad dos aristas etiquetadas cada una con un dígito.

Consideramos que esta introducción resulta suficiente para entender la aplicación que vamos a realizar de estos grafos. No obstante, para más información sobre ellos y sus aplicaciones, puede consultarse el trabajo de Eppstein ([6]).

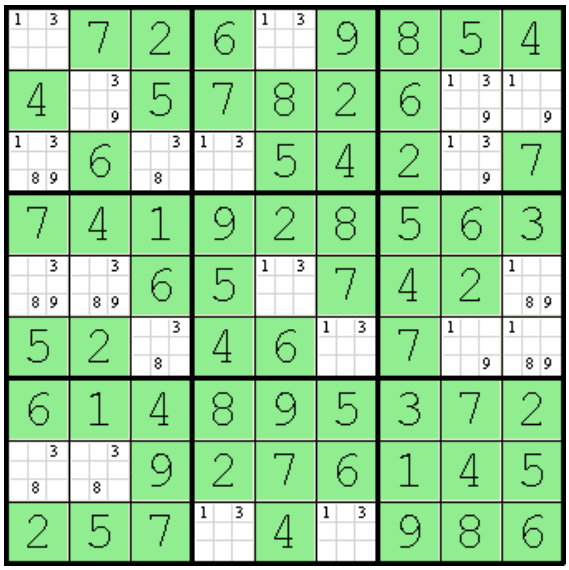

Figura 3.21: Tablero con casillas con sólo dos candidatos

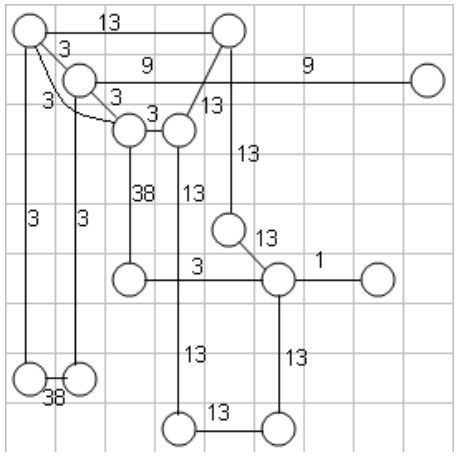

Figura 3.22: Ejemplo de creación de un *bivalue graph*

3.14.2. Aplicación de los *bivalue graphs* para encontrar patrones *XY-Chain*

La utilidad de estos grafos radica en que un camino sin repeticiones del grafo se puede traducir en una serie de deducciones lógicas consecutivas. Estas cadenas deductivas (*chains*), junto al hecho de que sólo se tratan casillas con dos candidatos (XY), explican el nombre que recibe este patrón.

Regla 1 Si el primer vértice de un camino sin repeticiones del grafo tuviera como valor la etiqueta de su arista, el vértice del otro extremo de la arista no puede tener ese valor.

Por lo tanto tendrá como valor a su otro candidato, que será el dígito de la siguiente arista, y así sucesivamente.

Regla 2 Si el grafo bivalorado contiene un ciclo en el que únicamente dos aristas consecutivas están etiquetadas con el mismo dígito d, la casilla que comparten ambas aristas no puede tener el valor final d, por lo que tendrá como valor final a su otro candidato.

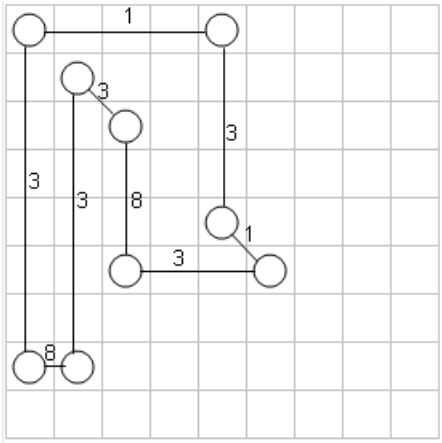

Figura 3.23: Ciclo detectado en el grafo anterior

Partiendo del grafo de la figura 3.22, la figura 3.23 muestra el subgrafo resultante de aislar aquellos vértices y aristas que forman el ciclo que cumple las condiciones requeridas por la regla 2.

En la figura 3.24 mostramos el patrón *XY-Chain* detectado sobre el tablero de la figura 3.21 y lo representamos de la siguiente manera:

- La casilla roja es en la que inciden las dos aristas marcadas con el mismo dígito (en este caso, es el 3).
- La casilla amarilla es una casilla intermedia del ciclo. A partir de ella se han creado dos caminos de distinto color, que se han de interpretar con la regla 1.

Por ejemplo, si la casilla amarilla $f1c5$ tiene por valor un 1, la casilla $f1c1$, conectada con ella mediante una arista etiquetada con 1, tendrá que tener su otro valor, un 3.

Razonando de esta manera, podemos formar dos cadenas deductivas a partir de la casilla amarilla $f1c5$:

- $f1c5 = 1 \Rightarrow f1c1 = 3 \Rightarrow f8c1 = 8 \Rightarrow f8c2 = 3 \Rightarrow f2c2 = 9$
- $f1c5 = 3 \Rightarrow f5c5 = 1 \Rightarrow f6c6 = 3 \Rightarrow f6c3 = 8 \Rightarrow f3c3 = 3 \Rightarrow f2c2 = 9$

- Como cualquiera de los dos valores que puede tomar la casilla amarilla fuerza a que la casilla roja tome el valor 9, podemos afirmar que ése será su valor definitivo.

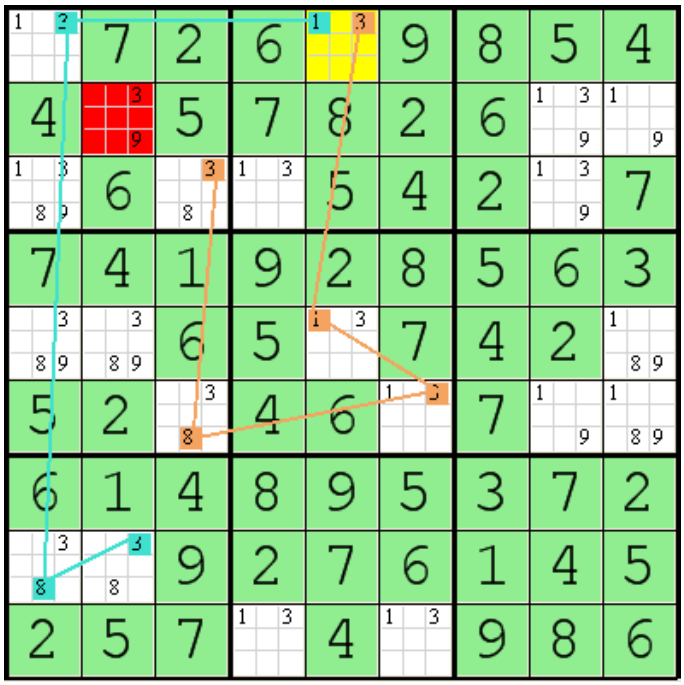

Figura 3.24: Detección de un *XY-Chain*

3.15. Patrón *Forcing Chains*

Este patrón es muy general y no hay consenso acerca de qué abarca y qué queda fuera de su alcance. Hay incluso quien considera que no es más que un disfraz para casos sencillos de prueba y error.

Sin embargo proporciona una gran potencia, por lo que lo hemos querido añadir a la colección de patrones implementados, limitándolo para que no ofrezca como pistas deducciones demasiado complicadas.

La versión implementada es la llamada *dual implication chains*. Consiste en partir de una casilla con sólo dos candidatos, y formar dos caminos con las implicaciones de colocar cada candidato como valor final. Asimismo hemos limitado el patrón para que sólo ofrezca pistas muy útiles, entendiendo por esto la colocación de un valor final en alguna casilla.

La forma más completa y eficiente de encontrar este patrón es utilizando dos tipos de grafos distintos, los *bivalue graphs* vistos en la sección anterior y los *bilocation graphs*, que introduciremos a continuación. Como ya hemos visto, el primer tipo de grafos nos permite hacer deducciones de *Naked Singles*. El grafo que añadimos ahora complementará al anterior con sus deducciones de *Hidden Singles*.

Para detectarlos manualmente sería muy complicado usar estos grafos, por lo que lo mejor será elegir una casilla con dos candidatos, y formar dos caminos con las consecuencias que tendría poner uno de ellos como valor final. Si en alguna casilla los dos caminos deducen que debe ir el mismo valor, podremos ponerlo.

3.15.1. *Bilocation graphs*

Empezaremos repasando lo que son dos *celdas conjugadas* para un candidato n, término que ya utilizamos en los patrones de *Coloring*, y que es necesario comprender para crear estos grafos. Dos celdas son conjugadas para un candidato n, si:

- Comparten grupo.
- Son las únicas que contienen el candidato n en ese grupo.

Dado un tablero de sudoku parcialmente resuelto, como el que aparece en la figura 3.25, a partir de las casillas sin valor definitivo podemos construir un *bilocation graph* aplicando el siguiente proceso:

1. Crear un vértice por cada casilla sin valor definitivo.
2. Conectar dos vértices con una arista etiquetada con el valor n, si los dos vértices comparten grupo y representan a dos casillas conjugadas para el candidato n.

En la figura 3.26 se muestra el grafo bivalorado que se obtiene a partir del anterior tablero. Nuevamente, las aristas que aparentemente están etiquetadas con dos dígitos son en realidad dos aristas etiquetadas cada una con un dígito.

Si se desea profundizar más acerca de estos grafos puede consultarse el trabajo de Eppstein ([6]).

1 3	7	2	6	1 3	9	8	5	4
4	3 9	5	7	8	2	6	1 3 9	1 9
1 3 8 9	6	3 8	1 3	5	4	2	1 3 9	7
7	4	1	9	2	8	5	6	3
3 8 9	3 8 9	6	5	1 3	7	4	2	1 8 9
5	2	3 8	4	6	1 3	7	1 9	1 8 9
6	1	4	8	9	5	3	7	2
3 8	3 8	9	2	7	6	1	4	5
2	5	7	1 3	4	1 3	9	8	6

Figura 3.25: Tablero parcialmente resuelto

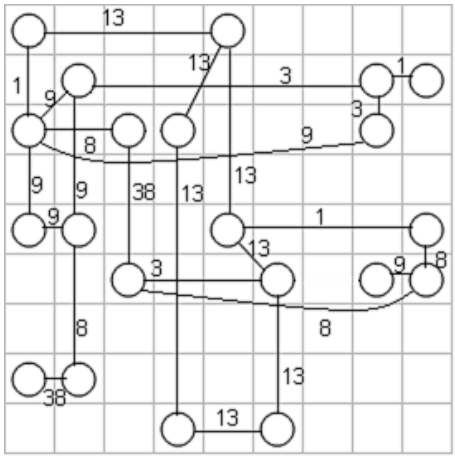

Figura 3.26: Ejemplo de creación de un *bilocation graph*

3.15.2. Aplicación de los *bilocation graphs* para encontrar patrones *Forcing Chains*

Un camino sin repeticiones de un *bilocation graph* se traduce directamente en una cadena de relaciones forzadas. Las deducciones son ligeramente distintas a las vistas para los *bivalue graphs*.

Regla 1 Si el primer vértice del camino sin repetición del grafo no tiene por valor el dígito de su arista, el otro extremo de la arista debe tener necesariamente por valor el dígito de la arista.

Para comprender esta regla, conviene recordar que en un *bilocation graph*, dos vértices están unidos mediante una arista si y sólo si se corresponden con las únicas dos casillas en el grupo que comparten un determinado candidato. Por tanto, si ninguno de los dos extremos tuviera por valor el dígito de la arista que los une, ninguno otra casilla podría tenerlo, llegando a una inconsistencia.

Esta regla es la que se aplicará en cada paso en el que se use el *bilocation graph*. Cuando se esté en una casilla c a la que se ha llegado por este método con una arista etiquetada con el dígito n, sabremos que el otro extremo tiene un valor distinto, por lo que la casilla c deberá tener el valor n.

Regla 2 Si el grafo tiene un ciclo sin repeticiones en el cual hay exactamente un par de aristas consecutivas marcadas con el mismo dígito d, entonces el vértice que comparten ambas aristas debe tener a d como valor definitivo.

La regla 2 nos permitiría encontrar cadenas sólo con deducciones del tipo *Hidden Single*. Sin embargo, no encontraría muchas cadenas que necesitan combinar pasos de este tipo con deducciones del tipo *Naked Single*. Combinar las deducciones de ambos grafos no era tarea fácil, y no hemos encontrado documentación al respecto.

Con este fin, ampliamos los *bivalue graphs* para que no sólo consideren casillas con exactamente dos candidatos, sino que tengan un vértice por cada casilla sin valor final, y aristas entre cada par de vértices correspondientes a casillas que compartan un candidato. La única restricción que imponemos es que uno de los extremos de cada arista debe tener exactamente dos candidatos.

Con estos dos grafos y la fusión de sus reglas 1 y 2 resultó el método que explicamos a continuación, y que proporciona unos resultados excelentes:

1. Se parte de una casilla con exactamente dos candidatos. En ella comienza la creación de un grafo mediante el método del *bivalue graph* (utilizando una arista del grafo *bivalue graph ampliado* que incida en el nodo correspondiente a esa casilla).

2. Cada casilla a la que se llega trata de seguir la formación del grafo de la siguiente manera:

   - Si se llegó por medio de un paso del *bivalue graph* y con el candidato n_1 sabemos que tenemos dos candidatos (n_1 y n_2) y que el valor final será n_2. Luego:
     - Podrá seguir por las aristas que indique el *bilocation graph* excepto por las que estén etiquetadas con n_2.
     - También podrá seguir por las aristas que indique el *bivalue graph* siempre que estén etiquetadas con n_2.
   - Si se llegó por medio de un paso del *bilocation graph* y con el candidato n_1, sabemos que el valor final será n_1. Luego:
     - Podrá seguir por medio de un paso del *bilocation graph* excepto por una arista etiquetada con n_1.
     - También podrá seguir por medio de un paso del *bivalue graph ampliado* siempre que la arista esté etiquetada con el dígito n_1.

3. Si llegamos a la casilla inicial y todavía no hay aristas consecutivas etiquetadas con el mismo dígito, habremos encontrado una cadena útil si:

   - Llegamos con un paso del *bivalue graph* con una arista etiquetada con el mismo dígito que el de la arista con la que comenzamos el ciclo.
   - Llegamos con un paso del *bilocation graph* con una arista etiquetada con un dígito distinto al de la arista con la que comenzamos el ciclo.

Al combinar ambos métodos es necesario restringir la arista por la que se sigue, como hemos visto. Se ve claramente con el ejemplo siguiente (la otra situación es análoga):

- Sea una casilla c con los candidatos n_1 y n_2, a la que hemos llegado mediante un paso del *bilocation graph* por una arista etiquetada con el dígito n_1.
- Por las reglas del grafo aplicado sabemos que c deberá tener el valor final n_1.
- Así, para continuar aplicando el método del *bilocation graph*, no podremos seguir por aristas etiquetadas con n_1, ya que el extremo siguiente concluiría que c tiene otro valor y se asignaría también el valor n_1 provocando una contradicción.

- Para continuar aplicando el método del *bivalue graph*, sólo podremos seguir por aristas etiquetadas con n_1, ya que si se siguiera por una arista etiquetada con otro dígito n_2, el siguiente vértice daría por hecho que c tiene el valor n_2, llegando también a una contradicción.

Para ilustrarlo mediante un ejemplo, consideremos el tablero parcialmente resuelto que aparecía en la figura 3.25. A partir de él, se construyen un *bilocation graph* y un *bivalue graph ampliado* como hemos indicado. Aplicando el método anterior, que combina ambos grafos, encontramos un ciclo tal y como se muestra en la figura 3.27.

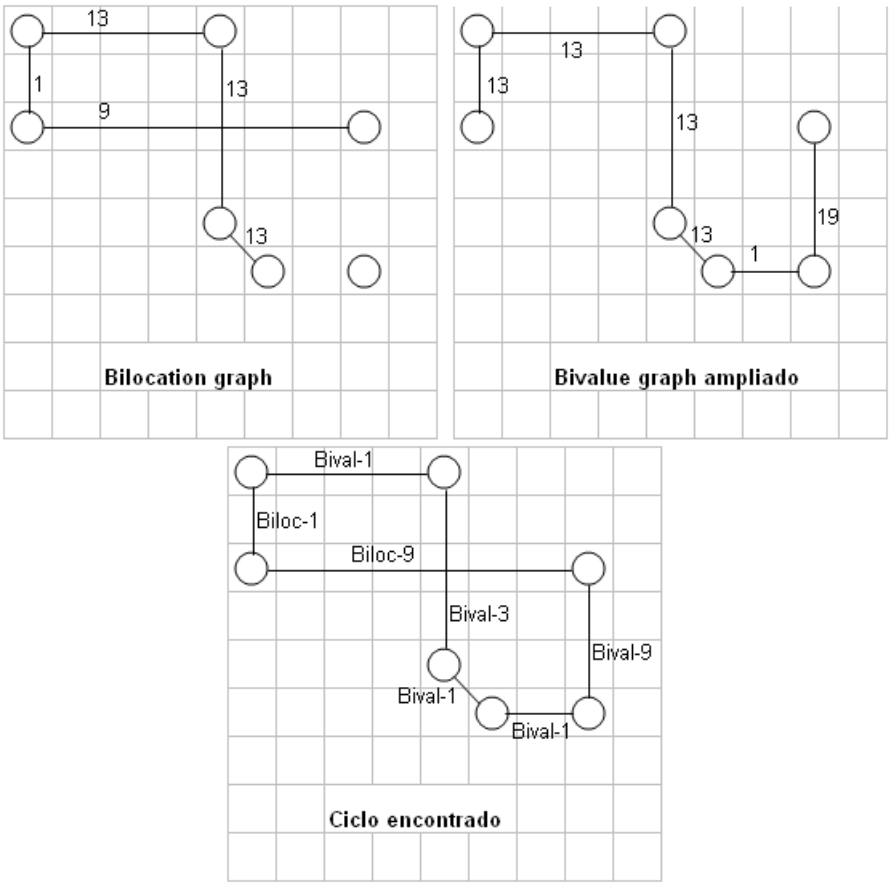

Figura 3.27: Ciclo encontrado, junto con la parte relevante de los dos grafos utilizados

En la figura 3.28 tenemos una representación gráfica de una detección del patrón sobre el tablero anterior. El modo de interpretarla es análogo al visto para el patrón *XY-Chains*:

- La casilla roja $f6c6$ es la que inicia el ciclo.

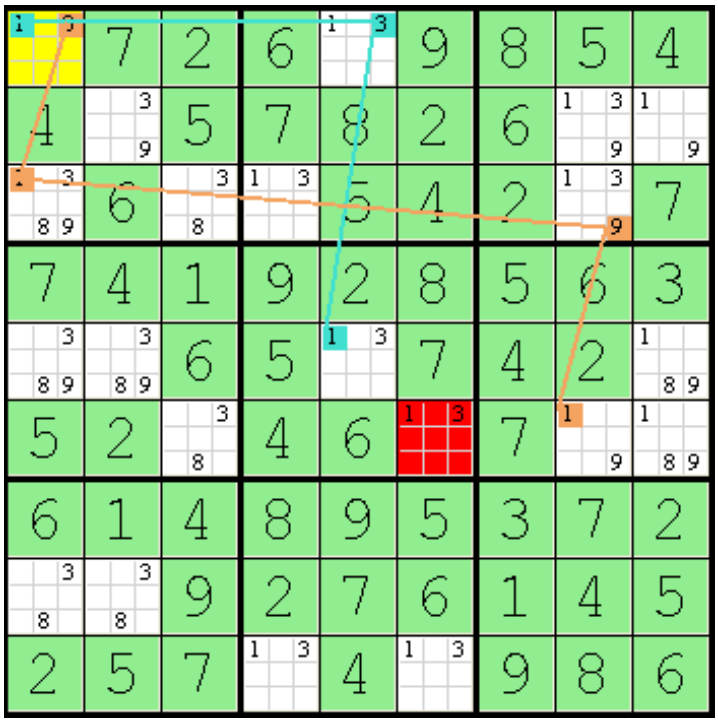

Figura 3.28: Representación gráfica de un *Forcing Chains*

- Se pasa por una arista del *bivalue graph* etiquetada con 1 a la casilla f5c5. Esto quiere decir que estamos suponiendo que la casilla roja tendrá el valor 1, y la casilla $f5c5$ tendrá su otro valor, el 3 (deducción *Naked Single*).

- Se pasa a la casilla $f1c5$ por una arista del *bivalue graph* etiquetada con 3. Por tanto esta casilla tendrá su otro valor, el 1 (*Naked Single*).

- Se pasa a la casilla $f1c1$ por una arista del *bivalue graph* etiquetada con 1. Por tanto esta casilla tendrá su otro valor, el 3 (*Naked Single*).

- Ahora pasamos a la casilla $f3c1$ por una arista del *bilocation graph* etiquetada con 1. Esto quiere decir que la anterior casilla no tiene el valor 1, por lo que $f3c1$ debe tener este valor (*Hidden Single*).

- Se pasa a la casilla $f3c8$ por una arista del *bilocation graph* etiquetada con 9. Por tanto esta casilla debe tener valor 9 (*Hidden Single*).

- Se pasa a la casilla $f6c8$ por una arista del *bivalue graph ampliado* etiquetada con 9. Por tanto esta casilla tendrá su otro valor, el 1.

  Éste es un ejemplo del tipo de deducciones que no podríamos haber hecho de no ampliar el *bivalue graph*.

- Llegamos a la casilla inicial $f6c6$ por una arista del *bivalue graph* etiquetada con 1. Es una de las condiciones de terminación del método.
- De la casilla roja salen dos aristas *bivalue* etiquetadas con 1, por tanto debe tener necesariamente como valor final su otro valor, el 3.

Capítulo 4

Resolución mediante prueba y error

Los algoritmos de *prueba y error* se pueden aplicar a la resolución de tableros de sudoku. Comparándolos con el método de resolución mediante patrones, descrito en el capítulo anterior, son más rápidos pero no siguen la lógica humana a la hora de resolverlos. Debido a esto, suelen ser útiles en la resolución de tableros de sudoku con otros fines distintos a ayudar al usuario a resolverlos. Un ejemplo de su uso es en la generación de tableros o en la comprobación de la unicidad de solución de los tableros que introduce el usuario.

En nuestra aplicación hemos encontrado dos tipos de algoritmos de prueba y error que son muy útiles para estos cometidos. Los describimos a continuación.

4.1. Algoritmo de vuelta atrás

Los algoritmos de vuelta atrás son algoritmos de búsqueda en un espacio de soluciones que no utilizan ninguna regla para hallar la solución, sino que se limitan a hacer pruebas de manera sistemática con todas las posibilidades hasta encontrar una que se ajusta. Si la búsqueda termina sin solución, es que ésta no existe. Es habitual que se divida la búsqueda en varias subtareas o búsquedas parciales, por lo que generalmente son algoritmos de naturaleza recursiva.

Tras cada prueba, si no se ha encontrado la solución se retrocede un paso y se prueba con otro valor distinto hasta agotar las posibilidades.

Puesto que a veces nos interesa conocer múltiples soluciones de un problema, estos algoritmos se pueden modificar fácilmente para obtener una única solución (si existe) o todas las soluciones posibles (si existe más de una) al problema dado.

Estos algoritmos se asemejan al recorrido en profundidad dentro de un

grafo, siendo cada subtarea un nodo del grafo. Este grafo se irá creando según avance el recorrido. A menudo dicho grafo es un árbol, o no contiene ciclos, es decir, al buscar una solución es, en general, imposible llegar a una misma solución x partiendo de dos subtareas distintas, o de la subtarea a es imposible llegar a la subtarea b y viceversa.

4.1.1. Aplicación a los sudokus

Vamos a construir un algoritmo de vuelta atrás que, dado un tablero de sudoku inicial, consiga:

1. Encontrar una solución de dicho tablero. En caso de no existir solución, también deberá indicarlo.

2. Contar el número de soluciones de dicho tablero. Puesto que un tablero inicial con pocas casillas puede tener del orden de millones de soluciones, y a nosotros sólo nos interesa ver que la solución del tablero es única, este algoritmo se detendrá cuando encuentre dos soluciones.

Para ello, en el primer caso, el algoritmo deberá devolver **falso** si el tablero no tiene solución, y **cierto**, junto con la solución, si ésta existe. En el segundo caso, se devolverá un entero $e \in \{0, \ldots, 2\}$, dependiendo del número de soluciones que tenga el tablero.

La solución del tablero la vamos a representar como una matriz de k filas por k columnas, siendo k el número de filas y columnas que tiene el tablero. Para los sudokus que estamos considerando, k toma el valor 9. En esta matriz apuntaremos el valor que asignamos a cada casilla. Debemos darnos cuenta de que inicialmente existen ya unos valores predeterminados, que son los valores iniciales del tablero, y que son inamovibles, luego habrá que anotarlos en la solución.

El espacio de búsqueda para este problema queda representado por el árbol de la figura 4.1. Vemos que inicialmente se le da el valor 1 a la primera casilla y se prueba a dar valores para la segunda casilla y, así sucesivamente hasta llegar a la casilla k^2-ésima. Por cada casilla podemos probar un rango de valores comprendidos en el intervalo $[1, \ldots, k]$.

Inmediatamente vemos que este árbol puede ser podado debido a las restricciones del problema. Así el nodo i-ésimo podrá ser podado si el número que se le ha asignado, ya se ha usado en una casilla de la misma fila, columna o bloque. Para saber si un número ya ha sido utilizado, vamos a utilizar unas estructuras auxiliares, que denominamos *marcadores*. Así, $F[i, num]$ va a ser una matriz de booleanos donde constará el valor **true** si el número num se ha insertado en la fila i, y **false** en caso contrario. De forma similar, funcionan $C[j, num]$ y $B[k, num]$ para columnas y bloques.

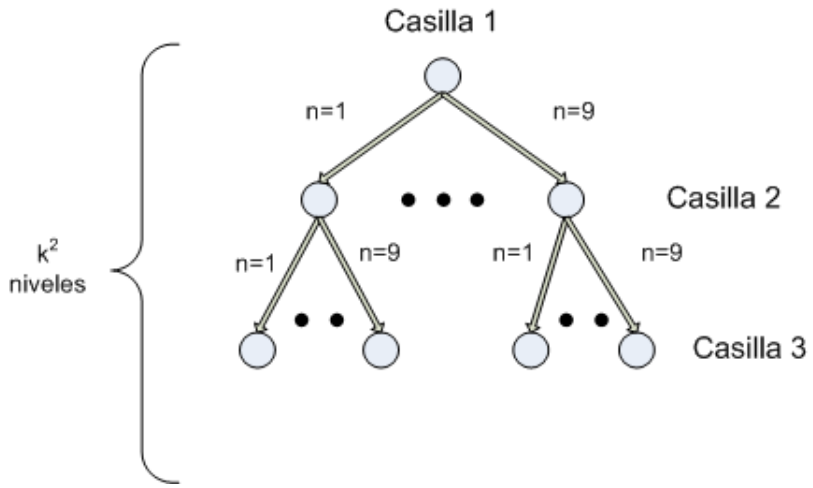

Figura 4.1: Espacio de búsqueda

4.1.2. Implementación

Con todo lo explicado anteriormente, ya sólo nos queda codificar el algoritmo de vuelta atrás que devuelve una solución de un tablero inicial dado, o indica si éste no tiene solución. Para la codificación vamos a utilizar pseudocódigo con el propósito de que todo sea entendido mejor.

```
proc sudoku(tableroAResolver[1..9, 1..9] de {0..9})
       dev {solucion de bool , sol[1..9, 1..9] de {1..9}}
    var
    C[1..9, 1..9] de bool
    F[1..9, 1..9] de bool
    B[1..9, 1..9] de bool

    para i:= 1 hasta 9 hacer
        para j := 1 hasta 9 hacer
      // si la casilla tiene un valor distinto de 0
      // entonces es que es una casilla que tiene valor
        valor := tableroAResolver[i,j]
        si valor<>0 entonces
          F[i, valor] := cierto;
          C[j, valor] := cierto;
          B[pasaABloque(i, j), valor] := cierto;
          sol[i, j]:=valor
        sino
            sol[i, j]:=0
        fsi
      fpara
```

```
    fpara
    solucion := falso;
    sudoku_va(sol, solucion, F, C, B, 1, 1))
fproc

proc pasaABloque(i,j de {1..9}) dev b de {1..9}
    casos
    i <= 3 && j <= 3 => b:=1;
    i <= 3 && j <= 6 => b:=4;
    i <= 3 && j <= 9 => b:=7;
    i <= 6 && j <= 3 => b:=2;
    i <= 6 && j <= 6 => b:=5;
    i <= 6 && j <= 9 => b:=8;
    i <= 9 && j <= 3 => b:=3;
    i <= 9 && j <= 6 => b:=6;
    i <= 9 && j <= 9 => b:=9;
    fcasos
froc

proc sudoku_va(E/S sol[1..9, 1..9] de {1..9},
               E/S solucion de bool,
               F[1..9, 1..9] de bool,
               C[1..9, 1..9] de bool,
               B[1..9, 1..9] de bool,
               i,j de {1..9})
numero := 1;
mientras numero <= 9 && not solucion hacer
    // casilla no asignada inicialmente
    si sol[i,j] = 0 entonces
    // si se cumplen las restricciones
        si not F[i, numero] && not C[j, numero]
               && not B[pasaABloque(i, j), numero])
        entonces
            // Asignamos y marcamos
            sol[i, j] := numero;
            F[i, numero] := cierto;
            C[j, numero] := cierto;
            B[pasaABloque(i, j), numero] := cierto;
            casos
            // llegamos al final, luego hemos
            // encontrado una solucion
                i==n && j==n => solucion := cierto;
            // hemos terminado la fila, pasamos a la siguiente
                i < n && j == n  =>
```

```
                sudoku_va(sol, solucion, F, C, B, i+1, 1);
            // seguimos a la siguiente casilla
                i <= n && j < n) =>
                   sudoku_va(sol, solucion, F, C, B, i, j+1);
            fcasos
            si not solucion entonces
                sol[i, j] := 0;
                F[i, numero] := falso;
                C[j, numero] := falso;
                B[pasaABloque(i, j), numero] := falso;
                numero++;
            fsi
        fsi
      sino
        casos
        // llegamos al final, luego hemos
        // encontrado una solucion
            i == n && j == n => solucion := cierto;
        // hemos terminado la fila, pasamos a la siguiente
            i < n && j == n =>
                sudoku_va(sol, solucion, F, C, B, i+1, 1);
        // seguimos a la siguiente casilla
            i <= n && j < n) =>
                 sudoku_va(sol, solucion, F, C, B, i ,j+1);
        fcasos
      fsi
fproc
```

4.2. Algoritmo *Dancing Links*

El algoritmo de *Dancing Links*, más comúnmente denominado *DLX*, es una técnica inventada por Hitotumatu y Noshita en 1979, pero popularizada por Donald Knuth para implementar eficientemente lo que Knuth llama *El Algoritmo X*, que es un algoritmo de tipo no-determinista y de prueba y error capaz de encontrar todas las posibles soluciones a un problema de recubrimiento.

El nombre de *Dancing Links*, enlaces que bailan, se debe a la forma en que el algoritmo trabaja, ya que las iteraciones del algoritmo hacen que los enlaces *bailen* de un nodo a otro.

A continuación, vamos a exponer tanto el *Algoritmo X* como su implementación usando *Dancing Links*. Estos métodos pueden ser consultados mas detalladamente en el trabajo de Knuth ([7]).

4.2.1. Algoritmo X

El *Algoritmo X* trabaja con una matriz de 1's y 0's. Esta matriz es conocida como la matriz *DLX* y se construye de la siguiente manera.

- Cada fila de la matriz se corresponde con un posible movimiento que se puede hacer en el problema en cuestión. Para el ejemplo del sudoku, un movimiento posible sería la asignación del número n en la posición (i, j).
- Cada una de las columnas de la matriz se corresponde con una restricción que cada solución debe satisfacer. En el ejemplo del sudoku una restricción sería que sólo puede asignarse un único número a cada casilla.
- Una celda de la matriz tendrá un 1 si el movimiento representado en la fila, cumple la restricción representada en la columna y tendrá un 0 en caso contrario.

Una vez que tenemos esta matriz, el problema de encontrar una posible solución consiste en encontrar un conjunto de movimientos, de tal forma que todas las restricciones se satisfagan. En términos de la matriz, el problema se reduce a encontrar un subconjunto de filas de la matriz, de tal forma que se consiga entre todas tener un exactamente un 1 en cada columna.

Veamos esto con un ejemplo. Sea M la matriz:

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Podemos ver, efectivamente, que para la matriz M existe un subconjunto de filas que, entre todas, contienen exactamente un 1 en cada columna. Estas filas son las número 1, 4 y 5.

Veamos ahora la implementación que Knuth expone en su trabajo sobre el *Algoritmo X*. Llamaremos A a la matriz *DLX* del problema.

```
si A está vacía entonces hemos encontramos la solución.
sino
    Elegimos una columna c (determinísticamente).
    Elegimos una fila f, tal que
            A[f, c] = 1 (no-determinísticamente).
    Incluimos a f en la solución parcial.
    para cada j tal que A[f, j] = 1,
        eliminamos la columna j de A;
```

```
            para cada i tal que A[i, j] = 1,
                eliminamos la fila i (no seleccionada) de A.
            fpara
        fpara
        Repetir el algoritmo para la matriz reducida.
    fsi
```

Cuando elegimos una columna donde sólo hay 0's, entonces hemos llegado a una situación donde no hay solución, y debemos realizar una vuelta atrás para seguir buscando una solución posible.

Para mejorar la eficiencia del algoritmo, se han de elegir primero aquellas columnas con menor número de 1's, pues así las búsquedas se realizan sobre los nodos con factores de ramificación más bajos.

Por último, veamos cómo funciona el algoritmo con un ejemplo. Vamos a buscar la solución para la siguiente matriz A:

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Vemos que el menor número de 1's en una columna es dos, y puesto que la columna 1 es la primera donde sólo hay dos 1's, vamos a empezar por ella. La fila 1 es la primera fila donde aparece un 1 en la columna 1, así que la introducimos en la solución parcial.

Ahora debemos buscar todas las columnas j tal que $A[1, j] = 1$, y eliminarlas. Luego podemos eliminar las columnas 1, 4 y 7.

El siguiente paso es buscar todas las filas i, tal que $A[i, j] = 1$, y eliminarlas. Esto quiere decir que debemos eliminar todas las filas que tengan 1's en las columnas antes eliminadas.

- Para la columna 1, podemos eliminar la fila 2. La fila 1 no puede ser eliminada, pues está seleccionada.
- Para la columna 4, podemos eliminar las filas 2 y 3.
- Para la columna 7, podemos eliminar las filas 2, 5 y 6.

Por lo tanto, se han de eliminar las filas número 2, 3, 5 y 6.

La matriz reducida tiene ahora este aspecto.

$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

La columna 1 no tiene ningún 1, así que hemos llegado a un punto donde no hay solución posible y debemos realizar la vuelta atrás. Para ello, restauramos la matriz anterior, y eliminamos la fila 1 de ella, pues hemos visto que no forma parte de la solución.

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Volvemos a elegir la columna número 1 puesto que es la columna con menor número de 1's, y la fila 1 es la fila donde aparece un 1 en la columna 1.

Eliminamos todas las columnas j donde aparezca un 1 en la primera fila, es decir, tal que $A[1, j] = 1$. Estas columnas son las número 1 y 4. Ahora debemos buscar todas las filas i, donde aparezca un 1 en las columnas anteriormente eliminadas, es decir, tal que $A[i, j] = 1$.

1. Para la columna 1, no se puede eliminar ninguna fila, pues la fila 1 está seleccionada.

2. Para la columna 4, podemos eliminar la fila 2.

La matriz reducida tiene este aspecto:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Puesto que todas las columnas tienen al menos un 1, podemos seguir con el proceso.

Ahora elegimos la columna 3, pues es la que menor número de 1's tiene, y para esta columna, la única fila con un 1 es la número 2.

Como antes, buscamos y eliminamos toda columna j tal que $A[2, j] = 1$. En este caso son las columnas 2, 3 y 4.

Ahora buscamos todas las filas i tal que $A[i, j] = 1$.

- Para la columna 2, podemos eliminar la fila 3, la fila 2 no pues está seleccionada

- Para la columna 3, no podemos eliminar ninguna fila.

- Para la columna 4, podemos eliminar la fila 3.

La nueva matriz reducida es:

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 1 \end{pmatrix}$$

Vemos que ambas columnas tienen un 1, luego podemos seguir con el proceso.

Elegimos la columna 1 porque es la primera en aparecer. Para esta columna, la fila 3 es la única que tiene un 1 en la columna 1.

Buscamos toda columna j tal que $A[3, j] = 1$. En este caso son las columnas 1 y 2. Puesto que debemos eliminar todas las columnas de la matriz, vemos que la matriz resultante queda vacía, luego hemos encontrado una solución.

Esta solución consta de todas las filas que hemos ido seleccionando durante todo el proceso, referidas a la matriz inicial, es decir, se trata de las filas número 2, 4 y 6 de A:

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

4.2.2. Aplicación a los Sudokus

Acabamos de explicar cómo funciona el *Algoritmo X* con un problema de recubrimiento general. No nos queda más que aplicar dicho método a la resolución de tableros de sudoku. Para ello, debemos saber cómo construir una matriz *DLX* para este tipo de problema.

Filas de la matriz *DLX*

Cada fila de la matriz *DLX* representa un posible movimiento. En el caso del sudoku, un movimiento consistirá en asignar un número a una casilla. Por ello, podemos representar a cada uno de los movimientos (o, lo que es lo mismo, cada una de las filas de la matriz) mediante una terna $\langle n, f, c\rangle$, donde n significa número, f fila y c columna, y todos ellos tienen un rango de valores del 1 al 9.

Por ejemplo, en esta matriz existirá una fila $\langle 2, 5, 4\rangle$ donde se asignará el valor 2, a la casilla $f4c5$.

Puesto que existen nueve dígitos posibles a colocar, y $9 \times 9 = 81$ casillas, existen un total de $9 \times 81 = 729$ posibles movimientos y, por tanto, 729 filas en la matriz *DLX*.

Columnas de la matriz *DLX*

Cada columna de la matriz *DLX* representa una restricción. En el caso del sudoku, existen cuatro tipos de restricciones que enumeramos a continuación:

1. Toda casilla (determinada por la fila y la columna en la que está situada) ha de tener un número asignado, y sólo uno: $\langle f, c\rangle$.

2. No puede haber dos números iguales en la misma fila: $\langle n, f\rangle$.

3. No puede haber dos números iguales en la misma columna: $\langle n, c\rangle$.

4. No puede haber dos números iguales en el mismo bloque: $\langle n, b\rangle$.

Por ejemplo, la restricción de tipo 1 $\langle 5, 7\rangle$, indica que la casilla $f5c7$ contiene exactamente un dígito. La columna de la matriz *DLX* para esta restricción tendrá un 1 para las filas $\langle 1, 5, 7\rangle$, $\langle 2, 5, 7\rangle$, ..., $\langle 9, 5, 7\rangle$. Se trata de todas las posibles asignaciones de números para esta casilla (del 1 al 9). El resto de la columna tendrá el valor 0. Puesto que hay nueve posibles filas y nueve posibles columnas, existen $9 \times 9 = 81$ columnas de tipo 1 en la matriz.

Una restricción de tipo 2 es $\langle 2, 5\rangle$, cuyo significado es que el número 2 debe aparecer sólo una vez en la fila 5. La columna de la matriz para esta restricción tendrá un 1 en las filas $\langle 2, 5, 1\rangle$, $\langle 2, 5, 2\rangle$, ..., $\langle 2, 5, 9\rangle$ que se corresponde con las posibles ubicaciones del número 2 en las celdas de la quinta fila: columnas del 1 al 9. Puesto que las soluciones válidas sólo seleccionarán una de estas nueve filas, el 2 únicamente aparecerá una vez en la quinta fila. Dado que hay nueve posibles números y nueve posibles filas, existen $9 \times 9 = 81$ restricciones del tipo 2.

De la misma manera se aplican las restricciones de tipo 3 (dígito-columna) y de tipo 4 (dígito-bloque), constando cada una de 81 elementos. Si contamos todas las restricciones de todos los tipos, tenemos $81 + 81 + 81 + 81 = 324$ restricciones y, por tanto, el mismo número de columnas en la matriz.

Cada fila de la matriz va a tener cuatro 1's: uno por cada restricción. Por ejemplo, la fila $\langle 2, 5, 7\rangle$, tiene 1's en las siguientes columnas:

1. En la restricción de tipo 1 $\langle 5, 7\rangle$ (número asignado a la casilla $f5c7$).

2. En la restricción de tipo 2 $\langle 2, 5\rangle$ (el número 2 sólo aparece una vez en la quinta fila).

3. En la restricción de tipo 3 $\langle 2, 7\rangle$ (el número 2 sólo aparece una vez en la séptima columna).

4. En la restricción de tipo 4 $\langle 2, 6\rangle$ (el número 2 sólo aparece una vez en la sexto bloque).

Por lo tanto, esta matriz de tamaño 729×324 tiene $729 \times 4 = 2916$ 1's y $729 \times 324 - 2916 = 254{,}988$ 0's, luego el número de 0's es mucho mayor que el de 1's, lo que lleva a pensar que parece más eficiente tener sólo nodos representando los 1's y trabajar con ellos.

Ejemplo de matriz *DLX* para un mini-sudoku

Vemos un ejemplo de matriz *DLX* para un mini-sudoku de tamaño 3×3 que, gracias a su tamaño reducido, permite mostrar la matriz resultante.

Así, generalizando esta representación, se facilitará la comprensión de la construcción de la matriz *DLX* que usaremos para nuestros sudokus.

Para este problema simplificado tendríamos las siguientes restricciones:

1. Toda casilla en una fila y columna ha de tener un número asignado, y sólo uno: $\langle f, c \rangle$.

2. No puede haber dos números iguales en la misma fila: $\langle f, n \rangle$.

3. No puede haber dos números iguales en la misma columna: $\langle c, n \rangle$.

La restricción que no permite que existan dos números iguales en el mismo bloque no tiene sentido en este ejemplo, dado que en este mini-sudoku no hay bloques. Puesto que tenemos nueve casillas y tres restricciones por casilla, la matriz *DLX* resultante va a tener un total de $9 \times 3 = 27$ columnas.

Ahora pensemos en los movimientos posibles. Tenemos nueve casillas y tres números, 1, 2 y 3, para situar en cada una de ellas. Por lo tanto, existirán $9 \times 3 = 27$ posibles movimientos o, lo que es equivalente, 27 filas en la matriz generada.

Para que todo quede más claro, numeraremos las filas utilizando la expresión $f,\ c = n$ donde $1 \leq f,\ c,\ n \leq 3$. La idea es que f representa la fila de la casilla seleccionada, c la columna y n el número que se asignaría a la casilla. De esta manera, queda claro que cada fila de la matriz *DLX* representa cada uno de los 3 posibles valores que pueden tener asignadas cada una de las 9 casillas.

De la misma manera, para favorecer la comprensión del ejemplo, numeraremos las columnas agrupándolas en tres grandes bloques, dependiendo de si representan restricciones de tipo 1, 2 o 3. Así, las columnas de tipo 1 vienen representadas por la tupla $\langle f, c \rangle$, cuyo primer elemento será $\langle 1, 1 \rangle$, el segundo el $\langle 1, 2 \rangle \ldots$ hasta llegar al último que será el $\langle 3, 3 \rangle \ldots$. Representan la restricción de que cada casilla ha de tener exactamente un número asignado. El resto de columnas se numerarán de forma análoga.

Una vez explicada la numeración, veamos cómo queda la matriz. En este ejemplo hemos omitido los 0's pues, como se explicó anteriormente, lo importante son los 1's, y sin ellos el ejemplo es más legible.

```
f,c=n        f,c                 f,n                 c,n

1,1=1   1 _ _ _ _ _ _ _ _  1 _ _ _ _ _ _ _ _  1 _ _ _ _ _ _ _ _
1,1=2   1 _ _ _ _ _ _ _ _  _ 1 _ _ _ _ _ _ _  _ 1 _ _ _ _ _ _ _
1,1=3   1 _ _ _ _ _ _ _ _  _ _ 1 _ _ _ _ _ _  _ _ 1 _ _ _ _ _ _
1,2=1   _ 1 _ _ _ _ _ _ _  1 _ _ _ _ _ _ _ _  _ _ _ 1 _ _ _ _ _
1,2=2   _ 1 _ _ _ _ _ _ _  _ 1 _ _ _ _ _ _ _  _ _ _ _ 1 _ _ _ _
1,2=3   _ 1 _ _ _ _ _ _ _  _ _ 1 _ _ _ _ _ _  _ _ _ _ _ 1 _ _ _
1,3=1   _ _ 1 _ _ _ _ _ _  1 _ _ _ _ _ _ _ _  _ _ _ _ _ _ 1 _ _
1,3=2   _ _ 1 _ _ _ _ _ _  _ 1 _ _ _ _ _ _ _  _ _ _ _ _ _ _ 1 _
1,3=3   _ _ 1 _ _ _ _ _ _  _ _ 1 _ _ _ _ _ _  _ _ _ _ _ _ _ _ 1
```

```
f,c=n        f,c                 f,n                 c,n

2,1=1   _ _ _ 1 _ _ _ _ _ _ _ _ 1 _ _ _ _ _ 1 _ _ _ _ _ _ _ _
2,1=2   _ _ _ 1 _ _ _ _ _ _ _ _ 1 _ _ _ _ _ _ 1 _ _ _ _ _ _ _
2,1=3   _ _ _ 1 _ _ _ _ _ _ _ _ 1 _ _ _ _ _ 1 _ _ _ _ _ _ _ _
2,2=1   _ _ _ _ 1 _ _ _ _ _ _ _ 1 _ _ _ _ _ _ _ 1 _ _ _ _ _ _
2,2=2   _ _ _ _ 1 _ _ _ _ _ _ _ 1 _ _ _ _ _ _ _ _ 1 _ _ _ _ _
2,2=3   _ _ _ _ 1 _ _ _ _ _ _ _ 1 _ _ _ _ _ _ _ _ _ 1 _ _ _ _
2,3=1   _ _ _ _ _ 1 _ _ _ _ _ _ 1 _ _ _ _ _ _ _ _ _ _ 1 _ _ _
2,3=2   _ _ _ _ _ 1 _ _ _ _ _ _ 1 _ _ _ _ _ _ _ _ _ _ _ 1 _ _
2,3=3   _ _ _ _ _ 1 _ _ _ _ _ _ 1 _ _ _ _ _ _ _ _ _ _ _ _ 1

3,1=1   _ _ _ _ _ _ 1 _ _ _ _ _ _ 1 _ _ 1 _ _ _ _ _ _ _ _ _
3,1=2   _ _ _ _ _ _ 1 _ _ _ _ _ _ _ 1 _ _ 1 _ _ _ _ _ _ _ _
3,1=3   _ _ _ _ _ _ 1 _ _ _ _ _ _ _ _ 1 _ _ 1 _ _ _ _ _ _ _
3,2=1   _ _ _ _ _ _ _ 1 _ _ _ _ _ _ 1 _ _ _ _ 1 _ _ _ _ _ _
3,2=2   _ _ _ _ _ _ _ 1 _ _ _ _ _ _ _ 1 _ _ _ _ 1 _ _ _ _ _
3,2=3   _ _ _ _ _ _ _ 1 _ _ _ _ _ _ _ _ 1 _ _ _ _ 1 _ _ _ _
3,3=1   _ _ _ _ _ _ _ _ 1 _ _ _ _ _ _ 1 _ _ _ _ _ _ _ 1 _ _
3,3=2   _ _ _ _ _ _ _ _ 1 _ _ _ _ _ _ _ 1 _ _ _ _ _ _ _ 1 _
3,3=3   _ _ _ _ _ _ _ _ 1 _ _ _ _ _ _ _ _ 1 _ _ _ _ _ _ _ 1
```

Para terminar el ejemplo, vamos a explicar cómo se rellena la matriz o, lo que es equivalente, dónde se colocan los 1's y por qué. Centrémonos en la decimosegunda fila, nombrada como $2,1 = 3$. Esta fila se corresponde con colocar un 3 en la casilla $f2c1$. Este posible movimiento tiene que cumplir tres restricciones:

- $\langle f, c \rangle = \langle 2, 1 \rangle$: la casilla $f2c1$ ha de tener exactamente un valor asignado.
- $\langle f, n \rangle = \langle 2, 3 \rangle$: la fila 2 ha de tener una única aparición del número 3.
- $\langle c, n \rangle = \langle 1, 3 \rangle$: la columna 1 ha de tener una única aparición del número 3.

Esa fila de la matriz, como todas, tiene que cumplir tres restricciones. Luego en esa fila debe aparecer exactamente un 1 por cada bloque de 9 columnas, es decir, exactamente 3 unos, cada uno de los cuales representa una de las restricciones que han de cumplirse.

Esta idea de representación forma parte de lo que hace Knuth para implementar el *Algoritmo X*. Su método de implementación recibe el nombre de *Dancing Links*, y pasamos a explicarlo a continuación.

4.2.3. *Dancing Links*

Como hemos comentado, es más eficiente utilizar una estructura de datos para representar exclusivamente los 1's de una matriz DLX determinada, y trabajar con ellos. Imaginemos que tenemos la siguiente matriz DLX:

$$\begin{pmatrix} a & b & c & d & e & f & g \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \tag{4.1}$$

La estructura que propone Knuth es una lista de nodos, donde cada nodo representa un 1 de la matriz. Cada nodo está conectado con otros nodos a través de enlaces, formando listas circulares doblemente enlazadas.

Para la anterior matriz, la estructura tendría el siguiente aspecto:

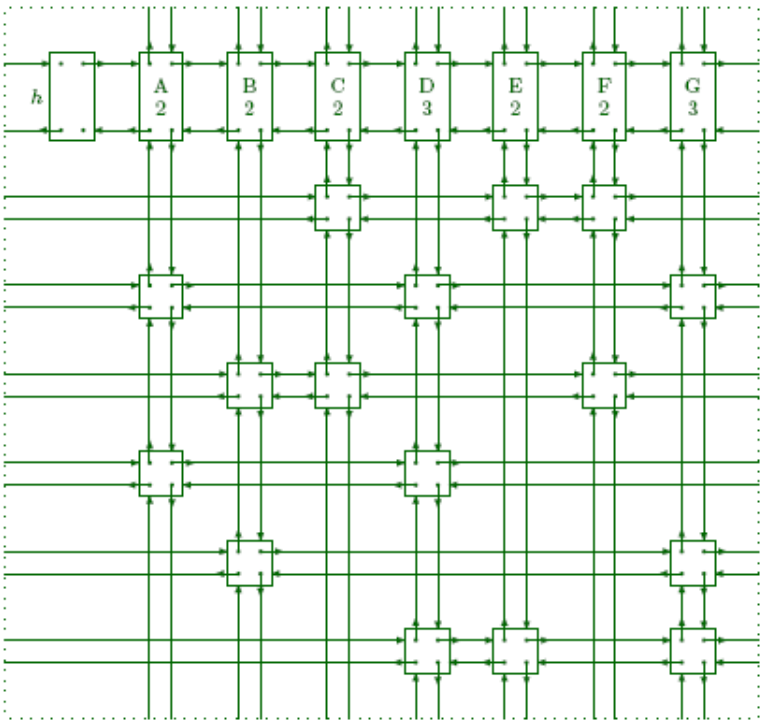

Figura 4.2: Estructura de datos usada en *Dancing Links*

A continuación, describimos cada uno de los elementos que componen la estructura representada en la figura 4.2. Para empezar, como se puede ver en la figura, existen dos tipos de nodos:

1. Los *data objects* son los nodos que representan a los 1's de la matriz.

   Cada uno de estos nodos está unido a otros nodos mediante los siguientes enlaces:

   - $L[x]$ y $R[x]$ apuntan al nodo anterior y posterior a x en la fila.
     Al ser una lista circular, el primero y el último se apuntarán el uno al otro.
   - $U[x]$ y $D[x]$ apuntan al nodo anterior y posterior a x en la columna.
     Cada columna tendrá un nodo especial, que será la cabecera de la columna, a la que apuntarán el primer y último elemento de la columna.
   - $C[x]$ apunta a la cabecera de la columna en la que está x.

2. Las cabeceras de las columnas se denominan *column objects*. Se trata de nodos especiales que sirven para poder avanzar por todas las columnas de la matriz que aún tienen 1's.

   La lista formada por los *column objects* está encabezada por un nodo especial denominado *list header*, que será el punto de entrada a la estructura de datos.

   Cada uno de estos nodos cabecera tiene los siguientes campos:

   - $L[x]$ y $R[x]$ apuntan a los nodos cabecera a la izquierda y a la derecha de x.
     La primera y la última cabecera apuntan al nodo especial llamado *list header*. Así se consigue conectar todas las columnas que necesitan ser cubiertas.
   - $U[x]$ y $D[x]$ apuntan al primer y último 1 de la columna.
   - $S[x]$ indica el número de 1's que quedan aún en la columna.
     Esto permitirá realizar la búsqueda de manera que se reduzca el factor de ramificación del árbol, eligiendo la columna que menos elementos tenga pendientes de fijar.
   - $N[x]$ guarda un nombre, para luego poder imprimir la solución. En este caso, se debe guardar el movimiento que representa dicha fila.

La eliminación de un nodo de la matriz así representada consiste simplemente en modificar los punteros de los nodos que apuntan a él, resultando en asignaciones como las siguientes:

$$L[R[x]] \leftarrow R[x]$$
$$R[L[x]] \leftarrow L[x]$$

Es decir, si $L[x]$ apunta al nodo v, y $R[x]$ apunta al nodo z, entonces la operación anterior se reduce a $R[v] = z$ y $L[z] = v$.

La extraordinaria ventaja que tiene esta representación es que, cuando sea necesario realizar una vuelta atrás, la reinserción en la estructura de un nodo anteriormente eliminado se reduce nuevamente a asignaciones como las siguientes:

$$L[R[x]] \leftarrow x$$
$$R[L[x]] \leftarrow x$$

Utilizando la terminología anterior, no hay más que hacer que $R[v] = x$ y $L[z] = x$. El punto está en que al no hacer operaciones de limpieza, el nodo x eliminado mantiene todos sus punteros, de manera que reinsertarlo tiene coste constante.

Como se ha podido observar, para poder eliminar filas y columnas, como se ha descrito en el *Algoritmo X*, se tienen que mover una gran cantidad de enlaces de unos nodos a otros, de ahí que parezca que los enlaces *bailen*.

Ya sólo nos queda ver cómo funciona este algoritmo. El algoritmo va a ser recursivo y se va a invocar con una llamada al procedimiento *search(k)*, siendo $k = 0$ inicialmente. Pasamos a describir este procedimiento:

```
proc search(k de entero)
/* h es el nodo cabecera de la lista formada por los
 * columns objects (la lista superior de la figura).
 * Esto significa que si el enlace derecho es el mismo
 * ya no quedan columnas con 1's y hemos terminado
 */
si R[h] = h entonces
    imprimir la solución y salir.
sino
    elegir un column object c (ver más abajo)
    cubrir la columna c (ver más abajo)
    // Recorremos todos los nodos de la columna c
    para cada r = D[c], D[D[c]]], . . . , mientras r <> c,
        // introducimos a r como parte de la solución.
        O[k] = r
        // recorremos la fila r.
        para cada j = R[r], R[R[r]], . . . , mientras j <> r,
            cubrir la columna j
        fpara
        // llamada recursiva con la matriz reducida
        search(k+1)
        // extraemos a r de la solución
        r = O[k]
        c = C[r]
```

```
        // deshacemos los pasos dados.
        para cada j = L[r], L[L[r]], . . . , mientras j <> r,
            descubrir la columna j (ver más abajo).
        fpara
    fpara
        descubrir la columna c y salir.
fsi
fproc
```

Para imprimir la solución, no tenemos más que imprimir cada una de las filas donde el elemento $O[i]$ está incluido. Para ello basta con realizar $N[C[O]]$, $N[C[R[O]]]$, $N[C[R[R[O]]]]$,... Una fila que pertenezca a la solución se imprimirá con el nombre de las columnas, donde cada uno de los elementos de esa fila están encuadradas.

Para la elección del *column object* podría elegirse la primera columna que contenga algún 1, simplemente realizando $c \leftarrow R[h]$. Sin embargo, como hemos puntualizado anteriormente, es preferible elegir las columnas que tengan un menor número de 1's. En este último caso, se debería realizar la asignación $s \leftarrow \infty$ y a continuación:

```
    para cada j = R[h], R[R[h]], . . . , mientras j <> h,
        si S[j] < s entonces
            c = j
            s = S[j]
        fsi
    fpara
```

La operación de cubrir una columna c es más interesante: elimina c de la *header list* y para cada nodo perteneciente a esta columna, también elimina las filas, a las que estos nodos pertenecen.

```
// eliminamos c de la header list (eliminamos la columna)
L[R[c]] = L[c] , R[L[c]] = R[c]
// recorremos la columna
para cada i= D[c], D[D[c]], . . . ,  mientras i <> c,
    // recorremos la fila donde esta i
    para cada j = R[i], R[R[i]], . . . , mientras j <> i,
         /* eliminamos cada nodo de la fila. Para ello
          * hacemos que el elemento anterior de la columna
          * donde i esta encuadrado apunte al elemento
          * posterior de la columna y viceversa */
         U[D[j]] = U[j], D[U[j]] = D[j]
         // también disminuye el número de 1's de cada columna
         S[C[j]] = S[C[j]] - 1
    fpara
fpara
```

Por último, sólo nos queda el paso de descubrir una columna. Para ello, no tenemos más que deshacer, por orden, todos los pasos dados en el proceso de cubrir.

```
para cada i = U[c], U[U[c]], . . . , mientras i <> c,
    para cada j = L[i], L[L[i]], . . . , while j <> i,
         S[C[j]] = S[j] + 1
         U[D[j]] = j
         D[U[j]] = j
    fpara
fpara
L[R[c]] = c, R[L[c]] = c
```

Una vez visto el algoritmo, vamos a estudiar su funcionamiento para la matriz 4.1, representada en la figura 4.2. Si aplicamos *search(0)* a la matriz 4.1, empezamos cubriendo la columna A, eliminando sus dos filas del resto de columnas. La estructura obtenida puede verse en la figura 4.3.

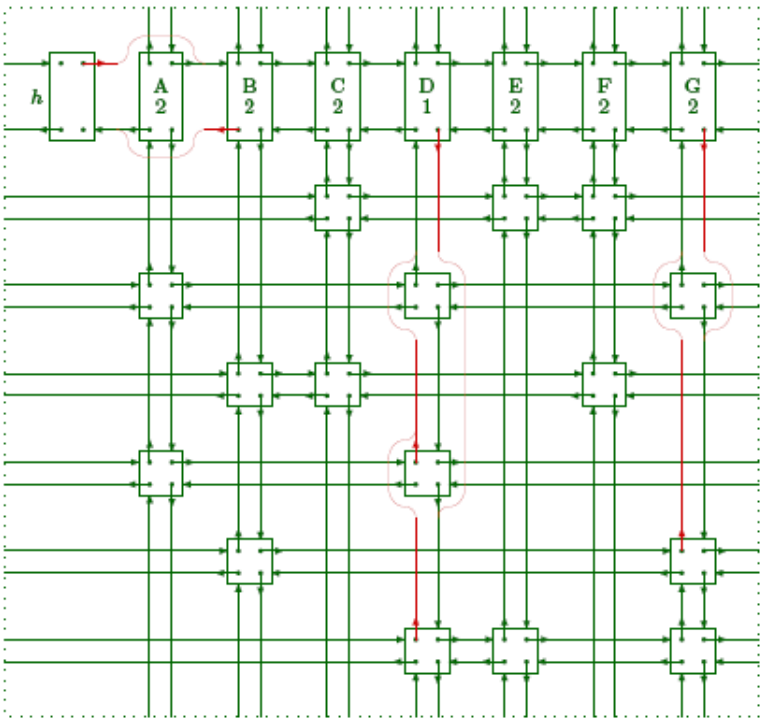

Figura 4.3: Estado de los enlaces después de cubrir A

Continuando con *search(0)*, cuando r apunta al elemento de A de la fila (A, D, G), tenemos que cubrir tanto la columna D como la columna G. La figura 4.4 muestra el resultado antes de entrar en *search(1)* y representa a

la matriz reducida:

$$\begin{pmatrix} B & C & E & F \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix}$$

Ahora *search(1)* cubrirá la columna B, y no habrá 1's en la columna E. Así que *search(2)* no encontrará nada, por lo que *search(1)* terminará sin encontrar soluciones y el estado de la figura 4.4 pasará a ser, otra vez, el de la figura 4.3 y volveremos a *search(0)* que avanzará r al elemento de la fila (A, D).

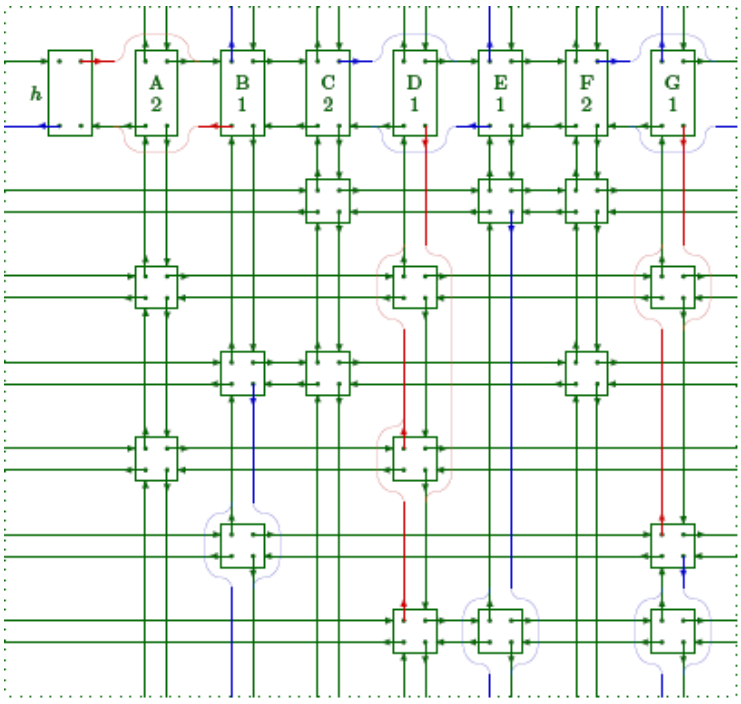

Figura 4.4: Estado de los enlaces después de cubrir A

Como se vio en el mismo ejemplo con el *Algoritmo X*, por este camino se consigue pronto obtener una solución que se imprimirá de la siguiente forma

$$A\ D$$
$$B\ G$$
$$C\ E\ F$$

si ignoramos el campo S a la hora de elegir columna o

$$A\ D$$
$$E\ F\ C$$
$$B\ G$$

si elegimos la columna con menor número de 1's.

Capítulo 5

Generación de tableros de Sudoku

La generación de tableros de Sudoku con cierto nivel de dificultad es una tarea costosa que puede suponer procesos muy costosos en tiempo si no se realiza de manera eficiente. Si bien hay infinidad de resolutores de Sudoku que siguen diversas técnicas, para la generación no existe tan amplia variedad de estrategias.

El mecanismo que se sigue para la creación de tableros con las características deseadas (nivel de dificultad, patrones de resolución que en él se requieran, etc.) es el mismo en todos los casos:

1. Se genera un tablero (con cualquiera de los métodos que explicaremos).
2. Se resuelve y en la resolución se estudia si en efecto cumple con lo deseado.

En caso negativo, se siguen generando más tableros y se repite la operación hasta que se halle lo buscado. Por este motivo, se hace esencial que el hecho de generar un tablero sea lo más rápido posible, puesto que habrá que generar un número indefinido de cuadrículas válidas antes de llegar a un tablero que satisfaga a lo requerido. También es importante que la resolución no ralentice el proceso, así que la clave para generar buenos tableros es utilizar algoritmos lo más optimizados posible.

En busca de la mejor opción, nos concentramos en estudiar los diferentes métodos de obtención de una cuadrícula de sudoku que cumpla las restricciones del juego. Para este aspecto en particular, sí hay diferentes aproximaciones.

Los métodos más populares de generación de tableros se limitan a elegir una casilla de las 81 posibles al azar (generación aleatoria de dos números del 1 al 9), y darle o quitarle un valor, dependiendo del tablero de partida para la generación. Este ciclo se repite tantas veces como sea necesario, hasta que se halla un tablero con una única solución o bien se llega a un estado en el que

no hay solución posible. En el primer caso, se da por generado el tablero; en el segundo, se puede desechar directamente o bien hacer *backtracking* para probar otra posibilidad.

Hemos realizado pruebas con dos alternativas de procedimientos basados en *backtracking*, que resultaron muy poco eficientes. Otro modo de generación que estudiamos utiliza el algoritmo *Dancing Links* presentado en el capítulo anterior, que fue popularizado por el profesor Donald E. Knuth como una herramienta de gran potencia para resolver problemas de recubrimiento. Este algoritmo es el más rápido para la resolución de tableros, y su uso dota de gran velocidad al método de generación. Es, con diferencia, la opción que mejores resultados proporciona y por tanto la hemos elegido para nuestro generador de sudokus.

A mayores, investigamos otros métodos de generación, como por ejemplo utilizar algún algoritmo genético. Esta idea se desechó porque los algoritmos genéticos no aseguran una solución cien por cien válida. Devuelven la opción más aproximada a la solución, y pierden eficacia si se limitan las funciones de mutación y cruce (esto se haría para hacer que los tableros manejados siempre cumpliesen las restricciones del juego). Si sumamos a todo ello que son algoritmos costosos en tiempo y en recursos de memoria, el uso de la *Programación Evolutiva* quedó totalmente descartado.

En los apartados siguientes veremos las formas principales que hemos estudiado para generar una cuadrícula válida. Posteriormente, se dedicará un apartado adicional a explicar las técnicas de valoración del nivel de dificultad que aplicamos a los tableros obtenidos.

5.1. Generación a partir de la cuadrícula vacía

En una primera versión, se parte de una cuadrícula totalmente vacía y se van escogiendo al azar casillas para rellenarlas con números aleatorios. El ciclo de decisión de casilla y búsqueda del valor aleatorio para la misma se repite hasta que el tablero sobre el que se han ido colocando números tiene una única solución.

El pseudo-código que mostramos a continuación representa esta versión del algoritmo de generación aleatoria de tableros. También se muestran las definiciones de los tipos Casilla y Sudoku, que se utilizarán en todos los algoritmos en pseudo-código que aparecen en este apartado.

```
Tipo Casilla es registro{
    Candidatos: array[1..9] de bool;
    Valor: int;
    }

Tipo Sudoku es array[1..9,1..9] de Casilla;
```

El siguiente algoritmo, `GeneraNuevoPartiendoDelVacio`, realiza un ciclo hasta que se obtiene un tablero válido que cumpla las expectativas fijadas de nivel de dificultad o cualquier condición que se quiera poner. Así, hay un bucle interno que llama a `CreaTablero` hasta que este algoritmo le devuelve una cuadrícula válida con una única solución. Una vez se sale de este bucle, se comprueba que el tablero nos sirve a través de la función `cumpleExigencias`. En caso de que el resultado de esta función sea *true*, se devuelve el tablero como el generado; en otro caso, se vuelve a empezar otro tablero desde cero.

```
Sudoku GeneraNuevoPartiendoDelVacio(){
    //inicialización de parámetros
        bool heTerminado := false; //aún no ha terminado
        int num:=0; //número de valores fijados inicialmente

  //Bucle de generación de tablero
  //con las características deseadas
  hacer{
    //Bucle de generación de tablero válido
    hacer{
        Sudoku s := new Sudoku();
        //inicialización del tablero:
        //todas las casillas sin valor y con
        //todos los candidatos posibles
        para (int x := 1; x <= 9; x++)
            para (int y := 1; y <= 9; y++)
                Casilla c := new Casilla();
                c.Candidatos[1..9] := true; //todos a true
                c.Valor := 0;
                s[x,y] := c;
            fpara
        fpara

        //llamada al algoritmo que va fijando valores
        CreaTablero(s,0,heTerminado);
    }mientras(!heTerminado);

    //comprobación de que cumple las exigencias de nivel de
    //dificultad, patrones que presenta etc.
    meGusta := cumpleExigencias(s);
  }mientras(!meGusta);

  return s;
}
```

`CreaTablero`, es el que realmente fija valores en la cuadrícula. Es un procedimiento recursivo que selecciona una casilla vacía de forma aleatoria en el tablero que se le pasa como parámetro de entrada/salida, y fija para ella un valor de entre los candidatos que tiene. Tras esta operación,

- si el tablero resulta no tener solución, se deshace el cambio y se sigue rellenando el tablero;
- si tiene una única solución, se devuelve como posible tablero (salida con valor de retorno *true* y `heTerminado` a *true* también);
- y si tiene más de una solución se continúa rellenando la cuadrícula mediante una llamada recursiva.

```
bool CreaTablero(E/S Sudoku tablero, E/S int cuantosNums,
            E/S bool heTerminado)
        {
            Sudoku tableroOriginal := tablero;

            //Elegir una casilla no rellena aleatoriamente
            int x, y;
            Casilla c;
            bool yaRelleno := true;
            hacer
            {
                x := aleatorio(1, 9);
                y := aleatorio(1, 9);
                c := s[x,y];
                yaRelleno := (c.Valor != 0);
            } mientras (yaRelleno);

            //Elegir un candidato posible para ella
            List<int> posibles := new List<int>();
            //crear la lista de posibles candidatos
            para (int i := 1; i <= 9; i++)
                si (c.Candidatos[i])
                    posibles.Add(i);
                fsi
            fpara
            si (posibles.Count = 0)
                heTerminado := false;
                return false;
            fsi
            int indice, candidato;
            hacer
```

```
        {
            indice:=aleatorio(1, posibles.Count);
            candidato := posibles[indice];
        } mientras (!c.Candidatos[candidato]);

        //Asignamos el valor a la casilla
        tablero[x,y].Valor := candidato;
        //Borramos "candidato" como posible candidato
        //   de la fila, columna y bloque
        tablero.LimpiaCandidatos();

        cuantosNums++;

        //Veamos cuantas soluciones tiene
        int numSoluciones := numSoluciones(tablero);
        casos
            numSoluciones = 0:
                tablero[x,y].Valor := 0;
                tablero := tableroOriginal;
                cuantosNums--;
                break;
            numSoluciones = 1:
                heTerminado := true;
                return true;
            numSoluciones = 2://tiene más de una
                break;
            default:
                break;
        fcasos

        si(CreaTablero(tablero,cuantosNums,heTerminado))
            return true;
        fsi

        tablero[x,y].Valor := 0;
        tablero := tableroOriginal;
        cuantosNums--;
        return false;
    }
```

5.2. Generación a partir de la cuadrícula completa

En la segunda versión de nuestros métodos de generación aleatoria, partimos de un tablero de sudoku relleno al azar completamente (por supuesto, cumpliendo las restricciones de no repetición de cifra en la fila, la columna y el bloque). Sobre él se aplica un ciclo de decisión de casilla al azar y eliminación de valor, es decir, se van eligiendo aleatoriamente qué casillas quedarán vacías en el tablero generado. Este ciclo se repite mientras que al quitar una casilla el tablero siga teniendo una sola solución. Cuando al quitar un valor el tablero se convierte en uno con más de una solución (o sin ella), se decide restablecer ese valor y se sigue ciclando.

En este caso, vamos a utilizado tres procedimientos. `GeneraRelleno()` cubre un tablero de sudoku completamente con valores, teniendo en cuenta que no se pueden repetir en la fila, la columna y el bloque. Para ello se sirve de `colocaCasilla()`, que es un algoritmo de vuelta atrás que va fijando valores mediante prueba y error. `GeneraNuevo()` es el procedimiento que realmente genera tableros con las condiciones impuestas. Consta de dos bucles anidados, similares a los de *GeneraNuevoPartiendoDelVacio*, ya que el externo cicla hasta que se cumplan las condiciones de dificultad impuestas al tablero, y el interno se encarga de asegurar que la cuadrícula tenga una única solución.

```
//genera un tablero de sudoku completamente relleno
Sudoku generaRelleno(){
        //inicialmente todas las casillas tienen a todos los
        //  números como candidatos
        //cPosibles: matriz 9*9 en la que cada posición es
        //  una lista de candidatos
        List<int>[,] cPosibles := new List<int>[1..9,1..9];
        para (int i := 0; i < 9; i++)
            para (int j := 0; j < 9; j++)
                List<int> candCasilla := new List<int>();
                for (int k := 1; k < 10; k++)
                    candCasilla.Add(k);
                cPosibles[i, j] := candCasilla;
            fpara
        fpara
        //solucion: almacen de los valores definitivos
        int[,] solucion := new int[9, 9];
        //rellenar tablero
        bool b;
        mientras (!b)
            b := colocaCasilla(cPosibles,solucion,0,0,r);
        fmientras
```

```
        //Tenemos una matriz válida. La convertimos a tablero.
        Sudoku nuevo := new SudokuDN();
        para (int i := 1; i < 10; i++)
            para (int j := 1; j < 10; j++)
                Casilla c := new Casilla();
                c.Candidatos[1..9] := false; //todos a false
                c.Valor := 0;
                nuevo[i,j] := c;
            fpara
        fpara

        return nuevo;
}


 //procedimiento recursivo que va rellenando las casillas
 //del tablero "solucion" utilizando backtracking
 //para encontrar un tablero relleno válido.
bool colocaCasilla(List<int>[,] cPosibles,
                   int[,] solucion,int i,int j,Random r){
    //hacemos una copia de los candidatos de la
    //casilla [i,j] por si hay que hacer backtracking
    int[] cProbados := new int[cPosibles[i, j].Count];
    cPosibles[i,j].CopyTo(cProbados);

    //mientras haya candidatos que probar
    //  para la casilla i,j
    mientras (cPosibles[i, j].Count != 0)
       int posVal:=aleatorio(0, cPosibles[i, j].Count);
       //pruebo valor para la casilla i,j
       int val := cPosibles[i, j][posVal];
       solucion[i, j] := val;
       //guardar los cambios hechos por si hay que
       //hacer backtracking
       List<int[]> celdasModif := new List<int[]>();

       //quitar el candidato de la columna
       para (int ii := 0; ii < 9; ii++)
           si (cPosibles[ii, j].Contains(val))
               celdasModif.Add(new int[2] { ii, j });
           cPosibles[ii, j].Remove(val);
       fpara
```

```
//quitar el candidato de la fila
para (int jj := 0; jj < 9; jj++)
    si (cPosibles[i, jj].Contains(val))
        celdasModif.Add(new int[2] { i, jj });
    cPosibles[i, jj].Remove(val);
fpara

//quitar el candidato del bloque correspondiente
//   a la casilla i,j
int numBloque := 0;
si (i < 3 && j < 3) numBloque := 1;
si (i < 3 && j >= 3 && j < 6) numBloque := 2;
si (i < 3 && j >= 6 && j < 9) numBloque := 3;
si (i >= 3 && i < 6 && j < 3) numBloque := 4;
si (i >= 3 && i < 6 && j >= 3 && j < 6) numBloque:=5;
si (i >= 3 && i < 6 && j >= 6 && j < 9) numBloque:=6;
si (i >= 6 && i < 9 && j < 3) numBloque := 7;
si (i >= 6 && i < 9 && j >= 6 && j < 9) numBloque:=8;
si (i >= 6 && i < 9 && j >= 6 && j < 9) numBloque:=9;
casos
    numBloque = 1:
        para (int ii := 0; ii < 3; ii++)
            para (int jj := 0; jj < 3; jj++)
            si (cPosibles[ii, jj].Contains(val))
               celdasModif.Add(new int[2] {ii, jj});
            cPosibles[ii, jj].Remove(val);
            fpara
        fpara
        break;
    numBloque = 2:
        para (int ii := 0; ii < 3; ii++)
            para (int jj := 3; jj < 6; jj++)
                si (cPosibles[ii, jj].Contains(val))
                celdasModif.Add(new int[2] {ii, jj});
                cPosibles[ii, jj].Remove(val);
            fpara
        fpara;
        break;
    numBloque = 3:
        para (int ii := 0; ii < 3; ii++)
            para (int jj := 6; jj < 9; jj++)
                si (cPosibles[ii, jj].Contains(val))
                celdasModif.Add(new int[2] {ii, jj});
                cPosibles[ii, jj].Remove(val);
```

```
            fpara
        fpara
        break;
    numBloque = 4:
        para (int ii := 3; ii < 6; ii++)
            para (int jj := 0; jj < 3; jj++)
                si (cPosibles[ii, jj].Contains(val))
                celdasModif.Add(new int[2] {ii, jj});
                cPosibles[ii, jj].Remove(val);
            fpara
        fpara
        break;
    numBloque = 5:
        para (int ii := 3; ii < 6; ii++)
            para (int jj := 3; jj < 6; jj++)
                si (cPosibles[ii, jj].Contains(val))
                celdasModif.Add(new int[2] {ii, jj});
                cPosibles[ii, jj].Remove(val);
            fpara
        fpara
        break;
    numBloque = 6:
        para (int ii := 3; ii < 6; ii++)
            para (int jj := 6; jj < 9; jj++)
                si (cPosibles[ii, jj].Contains(val))
                 celdasModif.Add(new int[2] {ii, jj});
                cPosibles[ii, jj].Remove(val);
            fpara
        fpara
        break;
    numBloque = 7:
        para (int ii := 6; ii < 9; ii++)
            para (int jj := 0; jj < 3; jj++)
                si (cPosibles[ii, jj].Contains(val))
                 celdasModif.Add(new int[2] {ii, jj});
                cPosibles[ii, jj].Remove(val);
            fpara
        fpara
        break;
    numBloque = 8:
        para (int ii := 6; ii < 9; ii++)
            para (int jj := 3; jj < 6; jj++)
                si (cPosibles[ii, jj].Contains(val))
                 celdasModif.Add(new int[2] {ii, jj});
```

```
                cPosibles[ii, jj].Remove(val);
            fpara
        fpara
        break;
    numBloque = 9:
        para (int ii := 6; ii < 9; ii++)
           para (int jj := 6; jj < 9; jj++)
                si (cPosibles[ii, jj].Contains(val))
                 celdasModif.Add(new int[2] {ii, jj});
                cPosibles[ii, jj].Remove(val);
            fpara
        fpara
        break;
    default:
        break;
fcasos

// llamada recursiva
int nuevoj := 0;
int nuevoi := 0;
si (j < 8)
   nuevoj := j + 1;
   nuevoi := i;
   si (colocaCasilla(cPosibles, solucion, nuevoi,
                     nuevoj, r))
       return true;
   sino
       solucion[i, j] := 0;
       //restablecer solucion[i,j] como candidato
       //en la fila, columna y bloque
       //excepto en la casilla i,j,
       //en la que ya hemos visto que no vale
       para (int k:=0; k<celdasModif.Count; k++)
           int[] celda := celdasModif[k];
           si(celda[0] != i || celda[1] != j)
             cPosibles[celda[0], celda[1]].Add(val);
       fpara
   fsi
sino
   nuevoj := 0;
   nuevoi := i + 1;
   si (nuevoi = 9)
       return true;
   si (colocaCasilla(cPosibles, solucion, nuevoi,
```

```
                            nuevoj, r))
                return true;
            sino
                solucion[i, j] := 0;
                //restablecer solucion[i,j] como candidato
                //en la fila, columna y bloque
                //excepto en la casilla i,j,
                //en la que ya hemos visto que no vale
                para (int k:=0; k<celdasModif.Count; k++)
                    int[] celda := celdasModif[k];
                    if(celda[0] != i || celda[1] != j)
                      cPosibles[celda[0], celda[1]].Add(val);
                fpara
            fsi
        fsi
    fmientras
        //si sale del bucle sin haber hecho return, es que
        //no ha encontrado solución en este subárbol.
        //Hay que forzar al backtracking del padre
        //restablecer los candidatos probados
        para (int c := 0; c < candidatosProbados.Length; c++)
            cPosibles[i, j].Add(cProbados[c]);
        return false;
    }


//devuelve un tablero válido para jugar que cumple los
//requisitos a tener en cuenta (dificultad etc.).
Sudoku GenerarNuevo(){
    bool haySolucion := false;
    Sudoku salida;//variable de salida

    mientras(!haySolucion)
    //partimos de un tablero relleno
      Sudoku nuevo := generaRelleno();

      bool terminado := false;
      //mientras no haya solución
      mientras (!terminado)

        //elegimos al azar una casilla para vaciar
        int i := aleatorio(1, 9);
        int j := aleatorio(1, 9);
```

```
        //vaciamos la casilla
        Casilla ca := new Casilla();
        ca.Candidatos[1..9] := true;//todos a true
        ca.Valor := 0;
        nuevo[i, j] := ca;

        //vemos cuántas soluciones tiene ahora
        int numSol := cuantasSoluciones(nuevo);
        si (numSol = 0)
            terminado := true; //se sale
            haySolucion := false;
        sino
            si (numSol = 1)
                si(cumpleExigencias(nuevo))
                        terminado := true;
                        haySolucion := true;
                        salida := nuevo;
                fsi
            sino
                //hemos quitado demasiadas casillas o
                //una no adecuada =>
                //devolvemos la casilla a su estado anterior
                Casilla c := new Casilla();
                c.Candidatos[1..9] := false;//todos a false
                c.Valor := solucion[i - 1, j - 1];
                nuevo[i, j] := c;
            fsi
        fsi
      fmientras
    fmientras
    return salida;
}
```

Este método era un poco más rápido que el anterior, por el sencillo hecho de que cada vez que se resolvía el tablero, éste tenía ya más valores fijos (se comienza con la cuadrícula totalmente rellena), así pues se ahorraba tiempo en la evaluación de la idoneidad del tablero generado. De todos modos, también es muy lento, y ambas opciones fueron desechadas.

5.3. Generación basada en *Dancing Links*

El método de generación basado en *Dancing Links* es el mejor de todos los que se han probado. Genera una cantidad muchísimo mayor de tableros que el resto de algoritmos, y en menos tiempo. Para la creación de tableros

sencillos es menos popular que las otras versiones, puesto que la comprensión de cómo resuelve tableros es más trabajosa, pero cuando se trata de generar tableros que verdaderamente reten a un jugador, no tiene rival.

El algoritmo de generación basado en *Dancing Links* es parecido a los ya explicados para la generación de cuadrículas, pero emplea Dancing Links para resolver tableros y un algoritmo de minimización para hacer que las cuadrículas obtenidas tengan el menor número posible de celdas rellenas por definición. En el capítulo de esta memoria dedicado a resolución viene explicado con detenimiento el algoritmo de resolución basado en *DLX*, y en este punto pasaremos a explicar cómo lo empleamos en el proceso de generación.

La implementación de *Dancing Links* que hemos utilizado se basa en la versión Java que se encuentra en `http://www.klepphelmer.com/Sudoku/`. La hemos adaptado a C# y hemos introducido algunos cambios, como el mecanismo de calibrado de la dificultad o transformaciones para hacer más comprensible el código. A continuación, pasaremos a explicar este método de generación e incluiremos pseudo-código como en los casos anteriores.

El primer algoritmo que veremos es el principal, `generaTablero()`. Se le proporciona un nivel de dificultad (`nivel`) y una semilla para generar números aleatorios. El nivel de dificultad se tiene que traducir a un rango, puesto que el algoritmo de resolución tenía su propio baremo y nosotros hemos empleado otro diferente. Para la traducción basta con guardar los rangos en una matriz, que en este caso se llama `RangosRating`. El método devuelve una cadena de texto, que será una representación del tablero resultado.

```
int[][] RangosRating := new int[][] {
   new int[]{5600, 999999},
   new int[]{5600,5900},
   new int[]{5800,9000},
   new int[]{6100,999999}
};

String generaTablero(int nivel, long initialSeed) {
  //Transformo el rating externo (1,2,3,4,5,6,7)
  //a un intervalo en el rango adecuado
  int minrating := RangosRating[0][0];
  int maxrating := RangosRating[0][1];
  si ((nivel>=0) && (nivel<RangosRating.Length))
     minrating := RangosRating[nivel][0];
     maxrating := RangosRating[nivel][1];
  fsi

  // Buscar un sudoku que cumpla el rango anterior
  // Para ello se generan tableros y se evalúan
```

```
  para (int v:=0; v<maxVueltas; v++)
     long semilla;
     si(initialSeed = 0)
        //inicializar semilla con un aleatorio
        semilla := aleat();
     sino
        semilla := initialSeed;
     fsi
     //numTableros: número de tableros que generará
     //  generadorDLX.generaTableros( , )
     int numTableros := 10;
     String[] tableros :=
             generadorDLX.generaTableros(semilla,numTableros);

     para (int i := 0; i < numTableros; i++)
        int rating;
        int numSol;
        this.resuelveYClasifica(tableros[i].Replace("\n", ""),
                                numSol, rating);
        if (rating >= minrating && rating <= maxrating)
            return tableros[i];
     fpara
  fpara
  return "";
}

String resuelveYClasifica(String tableroAResolver,
                          S int numSoluciones,
                          S int rating)
{
    //transformamos de formato al exigido por el resolutor
    String copiaLocal = tableroAResolver;
    si (copiaLocal.IndexOf("\n") > 0)
        copiaLocal.Replace("\n", "");

    return resolutorDLX.solve(copiaLocal,
                              S numSoluciones,
                              S rating);
}
```

Es un procedimiento sencillo, que llama al método `generaTableros()` un número de veces entre 1 y un máximo fijado. Dicho método devuelve una cantidad de tableros generados dependiendo del número que le indiquemos como parámetro. Para cada tablero devuelto, se llama al método que calibra

su dificultad, y si ésta está en el rango adecuado, se devuelve el tablero.

Hay que destacar que el método `resolutorDLX.solve()` que se encarga de clasificar a los tableros generados hace una clasificación aproximada basándose en cuánto trabajo realiza DLX para resolverlos, pero nosotros hemos implementado nuestro propio método de calibración de dificultad, que se ayuda de éste, pero se basa en qué patrones de resolución se emplean.

```
String[] generaTableros(long Seed, int numTableros)
    {
    InitializeRandomNumberGenerator(Seed);

    si (numTableros <= 0)
        numTableros := 100;
    fsi
    //valores iniciales de los tableros
    String[] result := new String[numTableros];
    para (int i := 0; i < numTableros; i++)
        result[i] := "";
    fpara

    initialize();//realiza las inicializaciones oportunas

    int numTableroActual := -1;
    int casillaElegida := 81;

    //estado inicial
    EstadoGeneracion estado := EstadoGeneracion.Inicial;

    //bucle de generación, se basa en transición de estados
    mientras (estado != EstadoGeneracion.Fin)
        casos
            //Inicial: Se encarga de ver si ya
            //tenemos suficientes tableros creados
            estado = EstadoGeneracion.Inicial:
                numTableroActual++;
                si (numTableroActual >= numTableros)
                    estado := EstadoGeneracion.Fin;
                    break;
                fsi
                estado := EstadoGeneracion.PonACeros;
                break;
            //PonACeros: Inicializa el tablero
            //con todas las posiciones a cero.
            estado = EstadoGeneracion.PonACeros:
```

```
    para (int i := 1; i <= 81; i++)
        A[i] := 0;
    fpara
    estado := EstadoGeneracion.EligeCasilla;
    break;
//EligeCasilla: Busca aleatoriamente una casilla
//aleatoria que no tenga valor asignado
estado = EstadoGeneracion.EligeCasilla:
    int casillaElegida;
    hacer
    {
        casillaElegida := aleat(80);
        casillaElegida++;
    } mientras (A[casillaElegida] > 0);
    estado := EstadoGeneracion.PruebaValor;
    break;
//PruebaValor: Prueba a asignarle un valor a
//la casilla elegida en el estado anterior
estado =  EstadoGeneracion.PruebaValor:
    int valor;
    valor := aleat(8);
    valor++;
    //Asignamos el valor a la casilla elegida
    //en el estado anterior y vemos cuantas
    //soluciones tiene el nuevo tablero.
    A[casillaElegida] := valor;
    numSolutions := solve();
    //Si no tiene solución volvemos a vaciarla.
    si (numSolutions < 1)
        A[casillaElegida] := 0;
    fsi
    //Si no tiene solución única
    //volvemos a buscar otra casilla
    si (numSolutions != 1)
        estado := EstadoGeneracion.EligeCasilla;
        break;
    fsi

    estado := EstadoGeneracion.Minimiza;
    break;
//Minimiza: En este estado ya hemos encontrado
//un tablero de solución única,
//  y trataremos de minimizarlo
estado =  EstadoGeneracion.Minimiza:
```

```
                //Probamos a vaciar casillas aleatoriamente
                int valorAntiguo,x;
                para (int k := 1; k <= 81; k++)
                    //Generamos una posición de casilla
                    x := aleat(80);
                    x++;
                    //Y la vaciamos
                    valorAntiguo := A[x];
                    si(valorAntiguo!=0)
                        A[x] := 0;
                        if (solve() > 1)
                        //Si vaciarla hace que el tablero
                        //tenga más de una solución,
                        //vuelvo a ponerlo
                            A[x] := valorAntiguo;
                    fsi
                fpara

                para (int i := 1; i <= 81; i++)
                //Escribimos el resultado a un
                //formato que entienda dlx_solver
                    result[numTableroActual] :=
                        result[numTableroActual] + A[i];
                fpara

                estado := EstadoGeneracion.Inicial;
                break;
        fcasos
    fmientras
    return result;
}
```

`generaTableros()` es el verdadero creador de cuadrículas. Está implementado como una máquina de estados, cuyo diagrama se muestra en la figura 5.1.

En el estado `inicial` se comprueba si ya se han generado suficientes tableros, en cuyo caso se va al estado final. En otro caso, pasa al estado `PonACeros`, que inicializa vacío un tablero nuevo para generar. Tras `PonACeros`, viene `EligeCasilla`, que toma una casilla al azar de entre las vacías. A continuación se pasa a `PruebaValor`, que coloca un valor que encaje en la casilla. Si se obtiene un tablero sin solución se elimina el valor. En este caso, o si el tablero no tiene solución única, se vuelve a `EligeCasilla`; en otro caso, se pasa a minimizar. `Minimizar` es un estado que va borrando valores de las casillas mientras el tablero siga teniendo una solución úni-

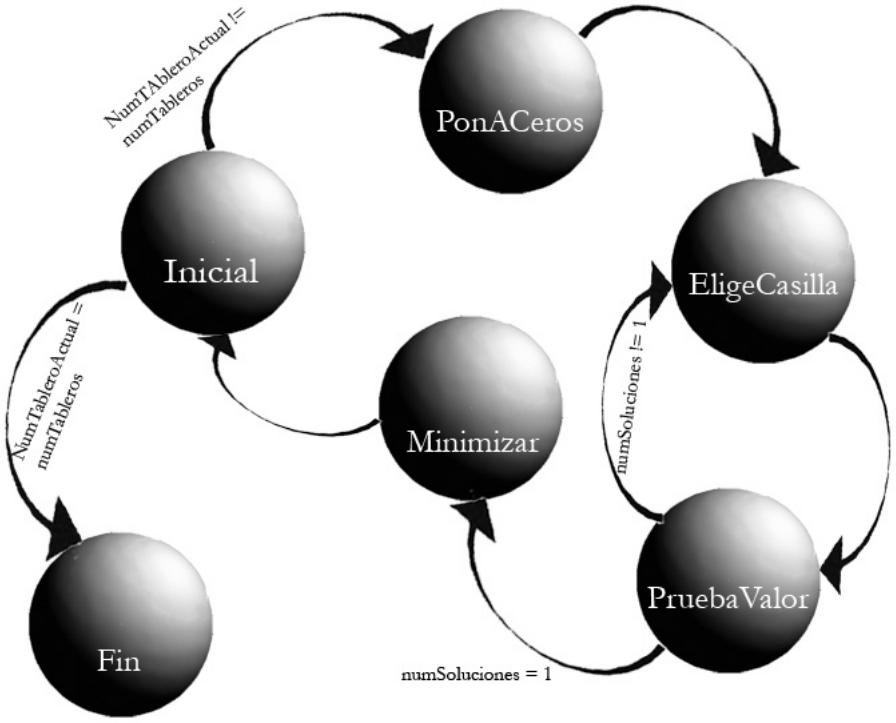

Figura 5.1: Diagrama de estados del generador de tableros

ca. Así se evita dar el tablero con más pasos resueltos de los estrictamente necesarios. Tras la minimización, se pasa al estado `inicial` para seguir generando más tableros o bien terminar (pasar al estado `Fin`). El estado `Fin` hace que se pare el bucle y se devuelvan los tableros.

5.4. Técnica de calibrado del nivel de dificultad

En el apartado anterior, vimos cómo el generador basado en *Dancing Links* genera un tablero en formato cadena de texto. A continuación, incluimos el procedimiento final que toma los tableros que va devolviendo dicho generador, transforma su formato al que entiende nuestro resolutor por patrones, y calibra su dificultad real. Para los algoritmos de generación que fueron desechados (los que no usan *DLX*) el trabajo de calibrado que se realiza en esta función es el que realizaría el método `cumpleExigencias()`.

El procedimiento tiene dos modos de uso:

- Si `esperoAlMejor = false` devuelvo el primer tablero encontrado de la dificultad deseada.
- Si `esperoAlMejor = true` ciclo durante `cuantosMinutos` minutos y devuelvo el mejor encontrado, siendo éste el que mayor número de patrones requiere del nivel deseado (teniendo en cuenta en menor medida los de niveles inferiores).

```
MotorDancingLinks dle := new MotorDancingLinks();

Sudoku GenerarNuevo(int nivel, bool esperoAlMejor,
                    float cuantosMinutos)
{
    //inicialización de variables
    //timer y transcurrido llevarán cuenta del tiempo empleado
    TimeSpan timer := TimeSpan.FromTicks(DateTime.Now.Ticks);
    TimeSpan transcurrido;
    Sudoku mejorEncontrado := null;
    float ratingMejorEncontrado := -1;
    float ratingSecundarioMejorEncontrado := -1;
    String tableroSS := "";
    Sudoku nuevo:=null;
    bool estoyConforme := false;
    bool puedoSeguir := true;

    long semilla := aleat();
    mientras (puedoSeguir || !estoyConforme)
        //1.-Genero un tablero de dificultad aproximada
```

```
casos
    nivel = 0: //muy facil
        tableroSS := dle.generaTablero(1, semilla);
        break;
    nivel = 1: //facil
        tableroSS := dle.generaTablero(1, semilla);
        break;
    nivel = 2: //medio
        tableroSS := dle.generaTablero(2, semilla);
        break;
    nivel = 3: //dificil
        tableroSS := dle.generaTablero(3, semilla);
        break;
    nivel = 4: //muy dificil
        tableroSS := dle.generaTablero(3, semilla);
        break;
    nivel = 5: //extremo
        tableroSS := dle.generaTablero(3, semilla);
        break;
fcasos

//2.-Transformación de formato
si (tableroSS = "")
//Si es vacío es que el generador no
//ha encontrado nada, lo vuelvo a llamar
    continue;
fsi

//transformación de formato, resultado en nuevo
formatoSStoFormatoDN(tableroSS,S nuevo);

//3.-Lo resuelvo con patrones y veo si la
// dificultad es la deseada
string texto;
int[] patronesUsados;
Sudoku clon := nuevo.Copia();
bool b := ResuelveTodo(clon, S texto,
                       S pUsados);
// "pUsados" es un array con el número
// de veces que se ha usado cada patrón
// para lograr resolver el tablero
si (b)//si lo ha resuelto
    //meGusta=true si no se usan patrones
    //más difíciles del nivel buscado
```

```
bool meGusta := true;
//meCuesta=true si se usan patrones
//del nivel buscado
bool meCuesta := false;
float cuantoCuesta := 0;
//meCuestaSecundario tendrá en cuenta el
//número de patrones aplicados de niveles
//inferiores al buscado
float meCuestaSecundario := 0;
int numPat := patronesUsados.Length;
casos
    nivel = 0: //muy facil
        para(int i:=0;meGusta && i<numPat; i++)
            si (pUsados[i] > 0)
                si (i != 0)
                    meGusta = false;
                fsi
            fsi
        fpara
        si (meGusta)
            //En este nivel, ningún tablero es
            //mejor que otro => fuerzo la salida
            TimeSpan  t := new TimeSpan(1,0,0)
            timer := timer.Subtract(t);
            meCuesta := true;
        fsi
            break;
    nivel = 1: //facil
       para(int i:=0; meGusta && i<numPat; i++)
          si (pUsados[i] > 0)
             si (i!=0 && i!=1)
                 meGusta := false;
             fsi
             si (i = 1)
               cuantoCuesta += pUsados[i];
               meCuesta := true;
             fsi
         fsi
       fpara
       break;
    nivel = 2: //medio
        para(int i:=0; meGusta && i<numPat; i++)
            si (pUsados[i] > 0)
                si (i != 0 && i != 1 &&
```

```
                    i != 2 && i != 3 &&
                    i != 4 && i != 5)
                    meGusta := false;
                fsi
                si (i = 2 || i = 3 ||
                    i = 4 || i = 5)
                    cuantoCuesta += pUsados[i];
                fsi
            fsi
        fpara
        meCuesta := (cuantoCuesta >= 2);
        break;
    nivel = 3: //dificil
        para(int i:=0; meGusta && i<numPat; i++)
            si (pUsados[i] > 0)
                si (i = 10 || i = 11 ||
                    i = 12 || i = 13 ||
                    i = 14 || i = 15)
                    meGusta := false;
                fsi
                si (i = 2 || i = 3 ||
                    i = 4 || i = 5)
                    meCuestaSecundario +=
                       (float)(pUsados[i] * 0.2);
                fsi
                si (i = 6 || i = 7 ||
                    i = 8 || i = 9)
                    cuantoCuesta += pUsados[i];
                fsi
            fsi
        fpara
        meCuesta := (cuantoCuesta > 0);
        break;
    nivel = 4: //muy dificil
        para(int i:=0; meGusta && i<numPat; i++)
            si (pUsados[i] > 0)
                si(i=13 || i=14 || i=15)
                    meGusta := false;
                fsi
                si(i=2 || i=3 || i=4 || i=5)
                    meCuestaSecundario +=
                       (float)(pUsados[i] * 0.1);
                fsi
                si (i=6 || i=7 || i=8 || i=9)
```

```
                        meCuestaSecundario +=
                            (float)(pUsados[i] * 0.3);
                    fsi
                    si (i=10 || i=11 || i=12)
                        cuantoCuesta += pUsados[i];
                    fsi
                fsi
            fpara
            meCuesta := (cuantoCuesta > 0);
            break;
        nivel = 5: //extremo
            para(int i:=0; meGusta && i<numPat; i++)
                si (pUsados[i] > 0)
                    si (i = 2 || i = 3 ||
                        i = 4 || i = 5)
                        meCuestaSecundario +=
                            (float)(pUsados[i] * 0.1);
                    fsi
                    si (i = 6 || i = 7 ||
                        i = 8 || i = 9)
                        meCuestaSecundario +=
                            (float)(pUsados[i] * 0.3);
                    fsi
                    si (i=10 || i=11 || i=12)
                        meCuestaSecundario +=
                            (float)(pUsados[i] * 0.5);
                    fsi
                    si (i=13 || i=14 || i=15)
                        cuantoCuesta += pUsados[i];
                    fsi
                fsi
            fpara
            meCuesta := (cuantoMeCuesta > 0);
            break;
    }

    estoyConforme := estoyConforme ||
                     (meGusta && meCuesta);
    si (meGusta && meCuesta &&
        ((ratingMejorEncontrado < cuantoCuesta) ||
         ((ratingMejorEncontrado = cuantoMeCuesta) &&
          (ratingSecundarioMejorEncontrado <
                              meCuestaSecundario)
         )
```

```
                )
               )
               si(!esperoAlMejor)
                 //fuerzo la salida
                 timer := timer.Subtract(new TimeSpan(1,0,0));
               fsi
               mejorEncontrado := nuevo;
               ratingMejorEncontrado := cuantoCuesta;
            fsi
        fsi
        transcurrido:=TimeSpan.FromTicks(DateTime.Now.Ticks);
        TimeSpan aux := transcurrido.Subtract(timer);
        si (aux.TotalSeconds >= 60*cuantosMinutos)
            puedoSeguir := false;
        fsi
    fmientras
    return mejorEncontrado;
}
```

El trabajo que realiza este algoritmo se concentra en un bucle que va realizando los siguientes pasos:

1. Genera un tablero utilizando el generador DLX. Le da un parámetro para la calibración de la dificultad que aproxime lo que queremos obtener (hemos decidido la correspondencia tras hacer pruebas con el motor DLX y ver qué patrones incluían sus niveles, aproximadamente).

2. Se convierte el tablero al formato que comprende nuestro resolutor.

3. Se resuelve el tablero con patrones, contabilizando cuáles se usan.

4. Se estudia qué patrones se han utilizado en la resolución. Si se ha usado alguno que quede en un nivel de dificultad superior al deseado, el tablero se desecha (es demasiado difícil). A la vez, se calcula cuántas veces se han usado los patrones que definen el nivel de dificultad, para comparar todos los que se ajusten y devolver el mejor.

5. Si el tablero obtenido es válido para el nivel de dificultad y buscábamos el primero encontrado salimos del bucle. Si buscábamos el mejor en un tiempo y es mejor que el que teníamos seleccionado hasta el momento, se marca como el mejor.

Tras ciclar este bucle durante `cuantosMinutos` minutos, se devuelve el mejor tablero obtenido.

Los patrones que definen los niveles de dificultad son los siguientes:

Muy fácil *Hidden Single*.

Fácil *Hidden Single* y *Naked Single*.

Medio Anteriores más *Naked/Hidden Pairs*, *Naked Triples* y *Locked Candidates*.

Difícil Anteriores más *X-Wing*, *Swordfish*, *Hidden Triples* y *Remote Pairs*.

Muy difícil Anteriores más *Colors*, *XY-Wing* y *XYZ-Wing*.

Extremo Anteriores más *Multicolors*, *XY-Chain* y *Forcing Chains*.

Para realizar esta clasificación nos hemos basado en la dificultad para una persona de entender los razonamientos que se encadenan para aplicar cada uno de los patrones, así como la dificultad de su detección.

5.5. Conclusiones

Tras estudiar varias formas de generar tableros de sudoku quedó claro que el bucle básico es equivalente para todos ellos:

1. Generar tablero
2. Comprobar si cumple las condiciones deseadas
3. Devolverlo en caso de éxito y volver a empezar en caso de fracaso

Para las tres opciones que se llegaron a implementar, esta estructura principal se mantiene, al igual que el principio básico de rellenado de un tablero: elegir casilla al azar y colocarle un valor al azar que encaje.

La diferencia principal está, por tanto, en las herramientas utilizadas para aceptar o desechar los tableros, para resolverlos y clasificarlos. En los dos primeros casos, se empleaban procedimientos de fuerza bruta con *backtracking* para contar el número de soluciones de un tablero, lo que los hacía verdaderamente lentos. Se llegaba a tiempos de proceso excesivamente grandes, cuestión que una aplicación que interactúa con el usuario no admite. El algoritmo de generación que emplea *Dancing Links* para estas resoluciones intermedias es considerablemente más rápido, además de incorporar un útil método de minimización de casillas cubiertas inicialmente que hace que los niveles de dificultad de los tableros se eleven considerablemente (con los otros dos métodos se solían resolver los tableros con *Naked* y *Hidden Singles*).

Tras sopesar las tres opciones, decidimos adoptar la mejor para la implementación de nuestro generador, y eliminamos las otras dos por su mal rendimiento y sus pobres resultados.

En cuanto a la calibración de la dificultad, gracias a que nuestro resolutor por patrones tenía implementado un recuento de los patrones aplicados, la tarea se realizó de forma bastante sencilla.

Capítulo 6

Implementación

Este capítulo contiene la descripción lógica de la aplicación mediante diagramas UML, así como algunos detalles de implementación e instalación.

6.1. Diagramas de casos de uso

En este apartado iremos mostrando cada uno de los diagramas de casos de uso del sistema. En la figura 6.1 puede verse el diagrama general y posteriormente se mostrará el diagrama de cada uno de los módulos.

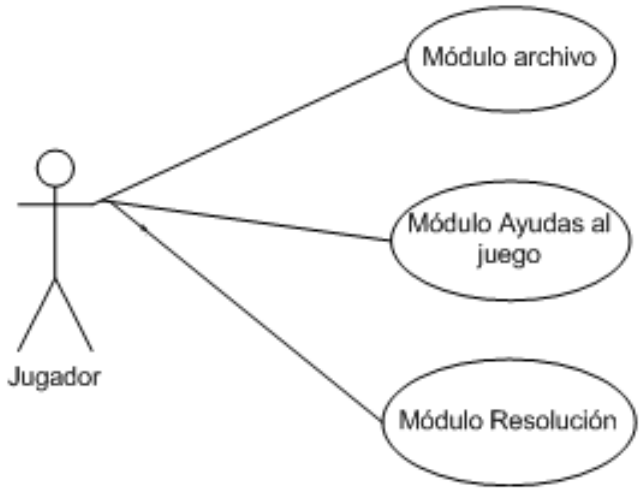

Figura 6.1: Diagrama general de casos de uso

El único usuario de la aplicación será el jugador, el cual podrá crear, abrir, guardar y resolver sudokus usando las diferentes ayudas que ofrece SuDoku JADE.

Módulo de archivo

Este módulo se encarga de gestionar todos los aspectos relacionados con la entrada y salida de tableros en el sistema. Su diagrama se muestra en la figura 6.2.

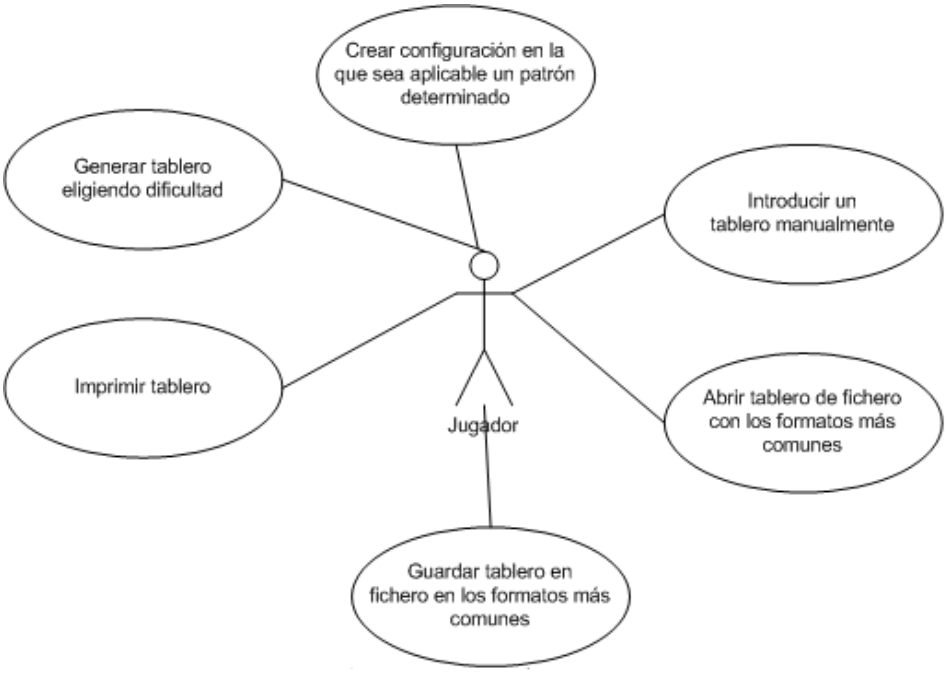

Figura 6.2: Diagrama de casos de uso del módulo archivo

Módulo de ayudas al juego

En este módulo se agrupan todas aquellas funciones que servirán de ayuda al jugador cuando intente resolver un tablero o aprender una nueva técnica. Son muy variadas pero todas tienen el mismo fin. El diagrama aparece en la figura 6.3.

Módulo de Resolución

Finalmente, este módulo contiene todas las funciones de resolución, ya sea paso a paso o completamente. En este último caso también se puede elegir el método de resolución, cada uno de los cuales aportará distintas ventajas e inconvenientes como ya hemos explicado al hablar de la funcionalidad del sistema. Su diagrama se muestra en la figura 6.4.

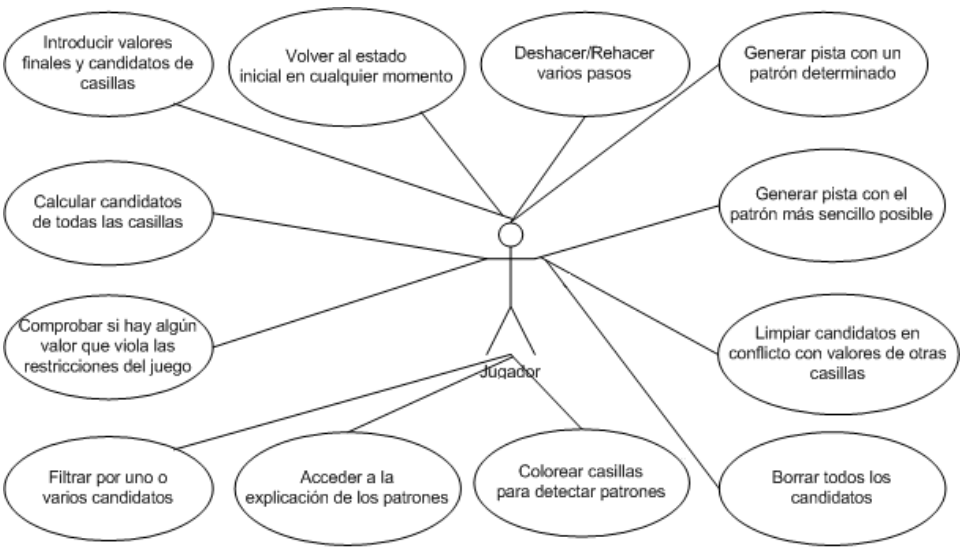

Figura 6.3: Diagrama de casos de uso del módulo de ayudas al juego

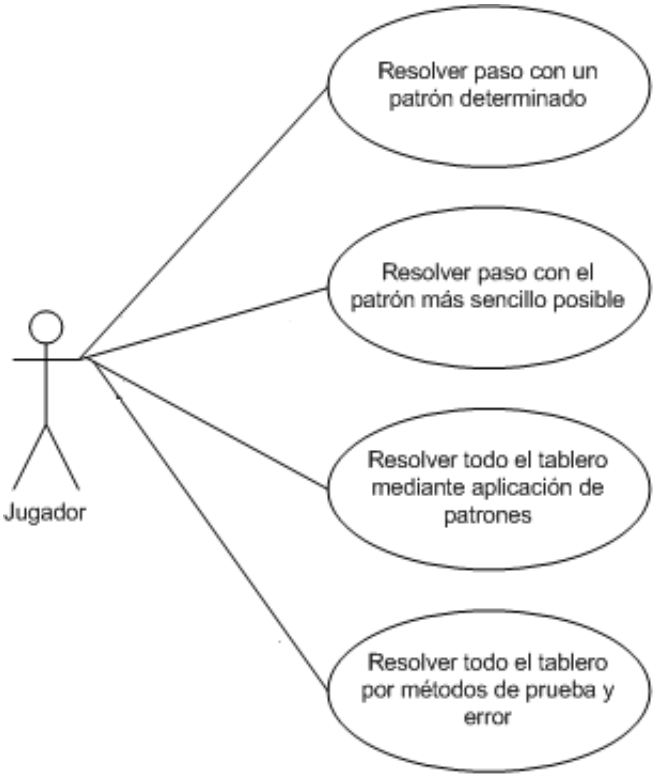

Figura 6.4: Diagrama de casos de uso del módulo de resolución

6.2. Diagramas de clases

Pasamos a presentar los diagramas de clases, donde se puede ver una visión estática de la arquitectura del sistema. Una primera visión general de las capas en las que se estructura la aplicación puede verse en la figura 6.5.

Figura 6.5: Diagrama de capas de la aplicación

Capa de presentación Es la interfaz de nuestro programa con el usuario (el jugador). Se trata una interfaz gráfica de *Windows Forms*.

Datos de negocio (DNs) Son las clases que contienen los datos de las entidades que tratamos: tableros, bloques, casillas, etcétera. Estos datos fluyen por todas las capas de la aplicación. El único cometido de estas clases es encapsular estos datos.

Lógica de negocio (LNs) Son las clases encargadas de efectuar las acciones necesarias tras cada petición procedente de la capa de presentación.

Fachada Las lógicas de negocio son diversas y la capa de presentación no tiene necesidad de conocer el ámbito de cada una, por lo que se incluye una fachada para centralizar las llamadas a la lógica y redirigir cada una a la clase correspondiente.

Para que quede legible el diagrama de clases, separaremos las clases en tres diagramas. Uno de los datos de negocio, otro de la capa de presentación y otro del resto.

6.2.1. Datos de negocio

En la figura 6.6 mostramos el diagrama de los datos de negocio. Se pueden observar dos bloques diferenciados: los tableros y los patrones.

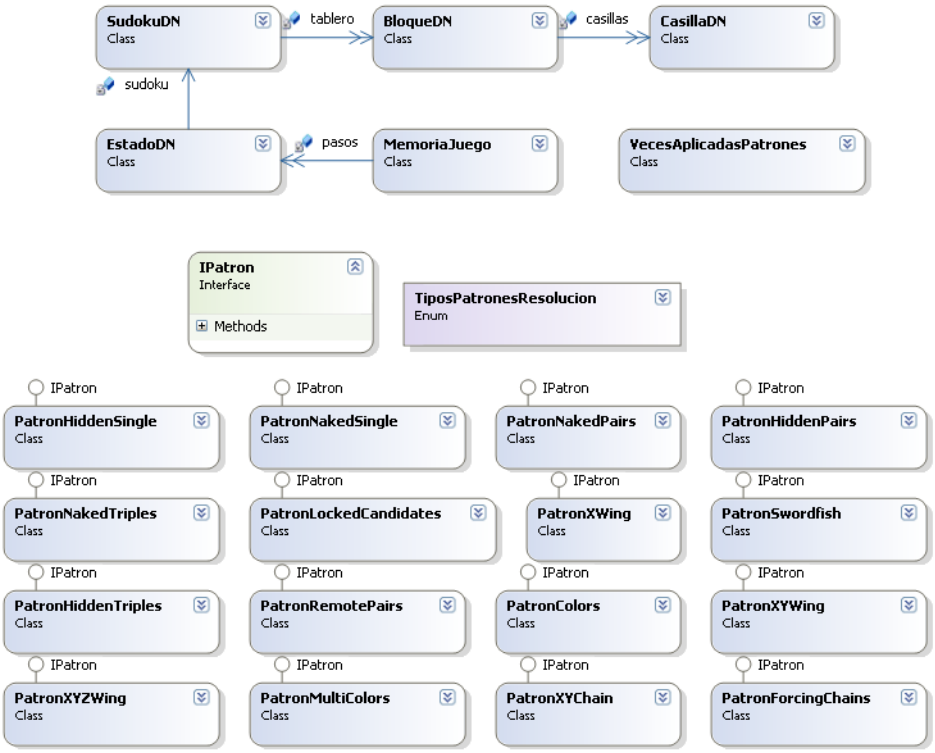

Figura 6.6: Diagrama de clases de los DNs

Tableros

Los tableros se componen de una colección de bloques, que a su vez se componen de una colección de casillas. Ha resultado adecuado representar los tableros de esta manera ya que gran número de operaciones, así como la validación de las reglas del juego se basan en este concepto. Para poder acceder directamente a casillas determinadas se crearon unos indexadores de `C#` en la clase `SudokuDN`. Cada clase tiene además de los datos, unos métodos para los que tienen toda la información necesaria y caen bajo su responsabilidad (patrón de diseño *GRASP Experto*, [11]).

La clase `MemoriaJuego` es la encargada de gestionar la cola circular creada para poder deshacer/rehacer pasos. Los datos necesarios de cada paso se encapsulan en la clase `EstadoDN`.

Patrones

El otro gran bloque lo forman los patrones. Se han implementado siguiendo el patrón de diseño *Strategy* ([12]). La interfaz `IPatron` define los siguientes métodos:

- `GeneraPista(...)`
- `ResuelvePaso(...)`

La lógica de negocio responsable de la resolución tiene un atributo de la clase `IPatron`, y dependiendo del patrón elegido por el jugador crea una instancia del patrón correspondiente, llamando a uno de estos dos métodos.

6.2.2. Capa de presentación

La estructuración de la capa de presentación se puede ver en la figura 6.7.

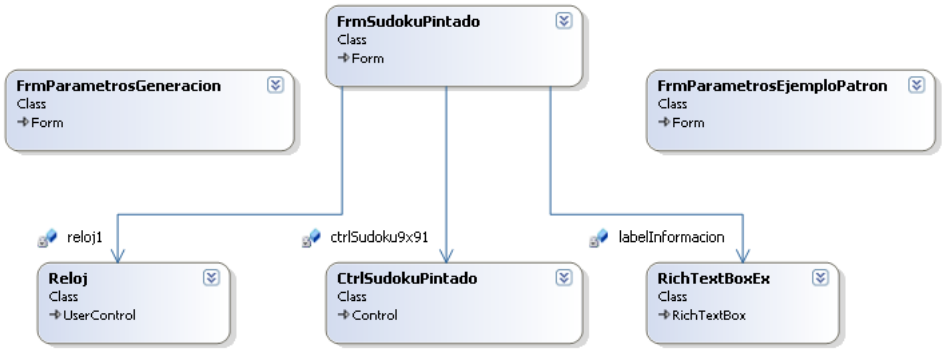

Figura 6.7: Diagrama de clases de la capa de presentación

El formulario principal es `FrmSudokuPintado`. Se ha aislado la representación gráfica del sudoku en el control `CtrlSudokuPintado`, así si en algún momento se quisiera cambiar por otra sería mucho más sencillo.

El formulario hace uso de otros dos controles:

- `Reloj` encapsula el comportamiento del reloj.
- `RichTextBoxEx` extiende la clase `RichTextBox` (del *framework* .NET) para añadirle la funcionalidad de introducir enlaces que no sean URLs. Este control lo hemos reutilizado de [16].

Por otra parte, se apoya en dos formularios secundarios para recoger los datos necesarios de la generación de tableros, y de la generación de ejemplos de patrones.

Dado que la funcionalidad del `FrmSudokuPintado` es muy grande, el código se ha dividido utilizando las *clases parciales* de .NET, que consisten en dividir una clase en distintos archivos, que el compilador une cuando realiza sus funciones.

6.2.3. Diagrama general de clases

La figura 6.8 muestra el diagrama general de clases. En él se observa que la clase `SudokuLN` realiza las tareas del patrón de diseño *Fachada* ([12]), y centraliza el acceso a las clases de lógica.

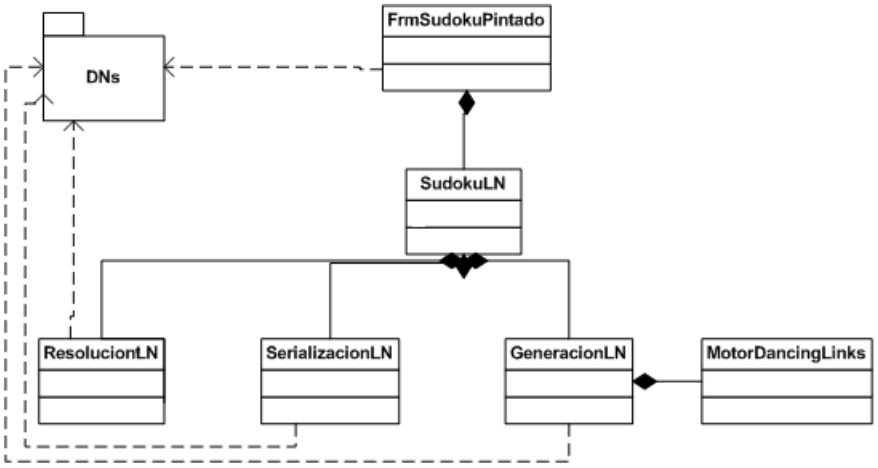

Figura 6.8: Diagrama general de clases

Estas clases se dividen según sus tareas:

- `ResolucionLN` es la responsable de todo lo relacionado con resolver un tablero o ayudar a hacerlo (generar pistas, calcular candidatos, ...).

- `SerializacionLN` es la encargada de abrir y guardar tableros a ficheros en todos los formatos soportados.
- `GeneracionLN` es la responsable de crear tableros y ejemplos de patrones, para lo que se apoya en la clase `MotorDancingLinks`, que se basa en un código traducido y modificado de la clase Java encontrada en [10].

6.3. Diagramas de secuencia

En este apartado mostramos los diagramas de secuencia para dos operaciones modelo. Comprendiendo estas dos se entenderán el resto de operaciones, ya que siguen una metodología análoga.

En la figura 6.9 se aprecia cómo el formulario sólo tiene que llamar a la clase `SudokuLN`, y ésta se encarga de hacer todas las operaciones necesarias:

- Abrir el fichero y realizar la deserialización.
- Comprobar que el tablero tiene solución.
- Actualizar el sistema.

Si la operación tiene éxito, se crea un estado con la configuración actual, y se inicializa la memoria con dicho estado.

En la figura 6.10 tenemos el modelo que siguen todas las operaciones de generar pistas, resolver pasos, resolver singles y resolver todo con patrones. Resolver todo por prueba y error, y calcular, limpiar y eliminar candidatos son también muy similares, excepto que no crean ningún objeto patrón.

- Primero se recupera el tablero que guarda el control, se valida y se llama a `GenerarPista` con ese tablero como parámetro.
- Se inicializa el sistema si no lo estaba.

  En otro caso se actualiza la información que guarda con la nueva.
- Se instancia un objeto de la clase patrón correspondiente, y se le pide que genere una pista aplicando las técnicas que tiene programadas.

6.4. Detalles de implementación e instalación

La aplicación SuDoku JADE está desarrollada sobre la plataforma .NET 2.0, por lo que para poder ejecutarla se hace necesario tener instalado el *.NET Framework Version 2.0 Redistributable Package* que se puede descargar de la siguiente dirección: `http://msdn.microsoft.com/netframework/downloads/updates/default.aspx`

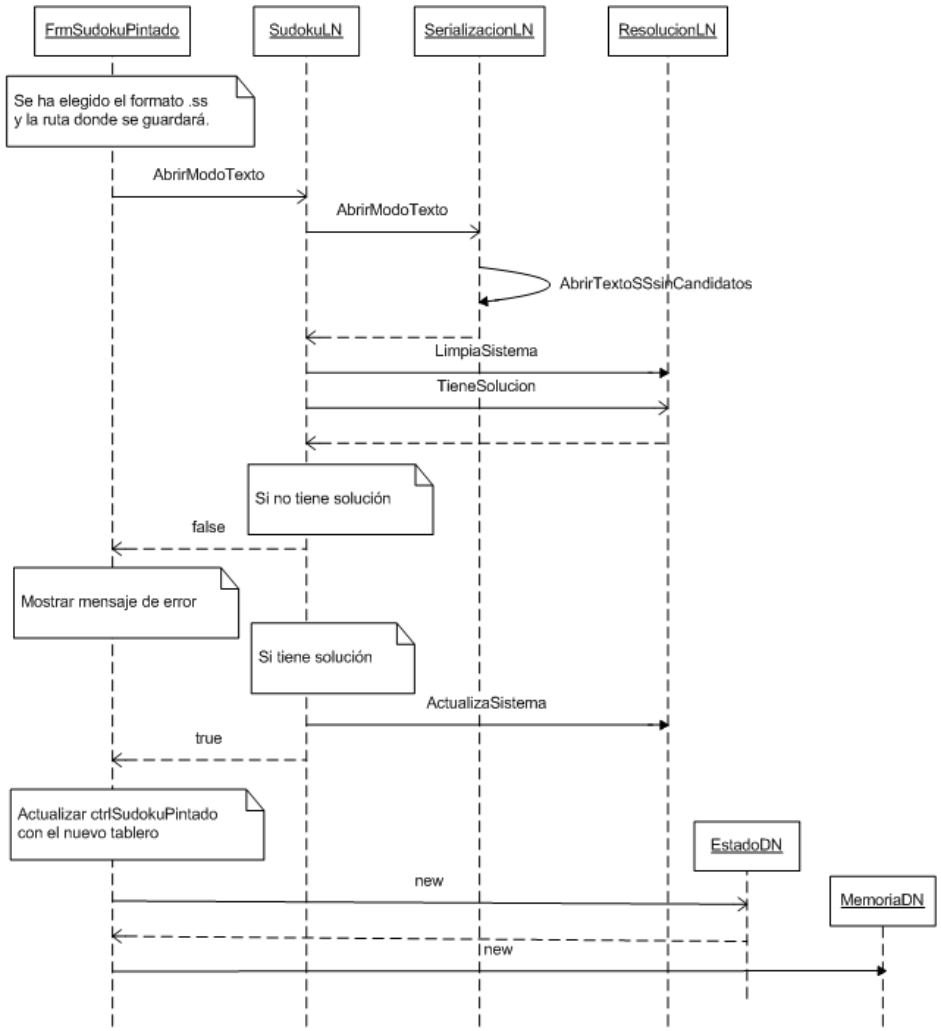

Figura 6.9: Diagrama de secuencia para la operación abrir (en formato *.ss*)

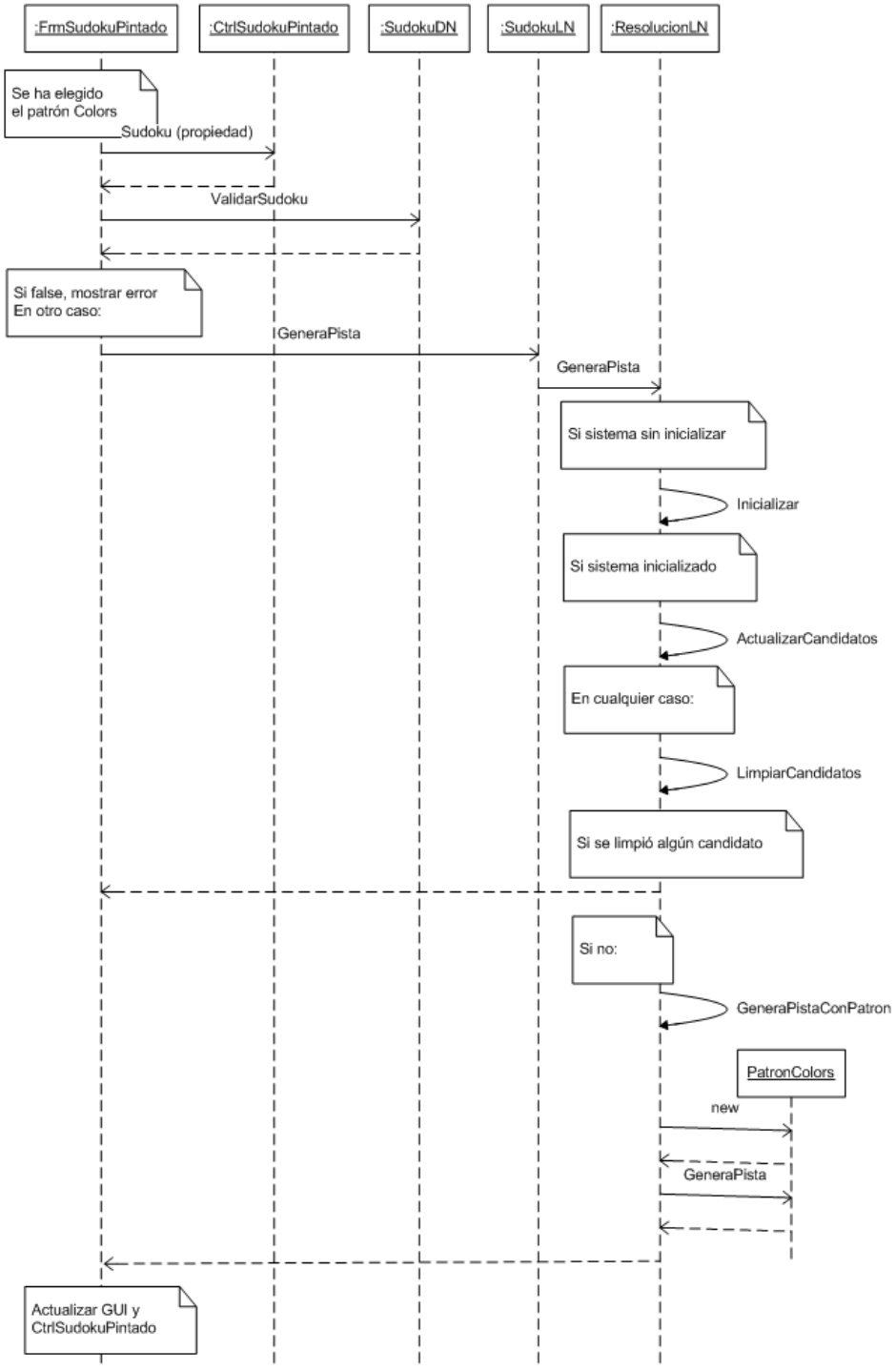

Figura 6.10: Diagrama de secuencia para la operación generar pista (con el patrón *Colors*)

Es una aplicación de escritorio realizada con *Windows Forms*, ya que queríamos ofrecer un grado de funcionalidad y usabilidad que no sería posible si estuviera implementada como una aplicación web.

Una vez tomada esta decisión, la elección de .NET frente a otras plataformas se debe a las facilidades que ésta ofrece para desarrollar aplicaciones competitivas.

El principal inconveniente es que se limita el sistema operativo *Windows* de *Microsoft*, a la espera de que el proyecto *Mono* ofrezca soporte para la versión 2.0 de .NET. Para obtener más información sobre el proyecto *Mono* y su estado actual, visítese la página web: `http://www.mono--project.com/Main_Page`

En cuanto a la estructuración del código, se trata de una solución .NET que consta de tres proyectos:

- `DatosNegocio` es el que contiene todas las clases vistas en el diagrama de la figura 6.6. Dentro de él se distingue un paquete `PatronesResolucion` que es donde se agrupan todos ellos.
- `GUI` contiene los formularios y controles de usuario de la figura 6.7.
- `LogicaNegocio` contiene la fachada y las LNs.

Para los proyectos .NET no hace falta ningún tipo de instalación. basta con copiar el ejecutable y las librerías necesarias. Para facilitar la instalación de nuestro sistema, hemos creado un proyecto de instalación que se encarga de realizar esta copia, comprobar si se tiene instalado el *framework* y crear accesos directos al programa.

El proyecto de instalación copia también un fichero de ayuda en formato *.chm* que se puede consultar desde la aplicación o de forma independiente, así como un conjunto de tableros de ejemplo de cada uno de los niveles de dificultad.

Capítulo 7

Conclusiones

7.1. Objetivos cumplidos

Hemos cubierto los objetivos principales que se plantearon al comienzo del proyecto. Los tres principales eran la resolución mediante la técnica de los patrones, la utilización de éstos para generar pistas y resolver pasos aislados mostrando al usuario el razonamiento seguido, y la generación de tableros de diferentes niveles de dificultad. Todos ellos han sido implementados con éxito.

Nos planteamos todas las ayudas posibles que podíamos ofrecer al usuario, y de ahí salió una parte importante de la funcionalidad como la posibilidad de coloreado de casillas, o la introducción, cálculo automático y filtrado de candidatos.

Por último añadimos funciones no imprescindibles para la resolución de sudokus pero que amplían las posibilidades del programa, como guardar, cargar e introducir manualmente tableros.

En la ayuda, que se puede consultar desde el sistema, se encuentra un manual de usuario de la aplicación, la explicación de todos los patrones así como la terminología necesaria para entenderlos.

Una vez cumplidos todos estos objetivos, nos dedicamos a buscar nuevas posibilidades que ampliaran la experiencia del usuario. Así surgieron ideas implementadas que se han demostrado muy útiles como imprimir los tableros en papel, acciones de deshacer y rehacer pasos, y generar configuraciones en las que se aplique un patrón determinado.

7.2. Principales problemas encontrados

7.2.1. Interfaz gráfica del tablero de Sudoku

Desde el principio, el objetivo era implementar una interfaz gráfica de usuario para el tablero que cumpliera varios requisitos:

- Tuviera una apariencia agradable.
- Permitiera al usuario redimensionar el tablero según sus preferencias.
- Ofreciera una forma sencilla y rápida de introducir valores, candidatos y colores.
- Presentara de forma lo más intuitiva posible los patrones cada vez que el sistema genera una pista o resuelve un paso.

Con este fin, aislamos la representación del tablero en un *control de usuario* independiente del formulario, lo que nos permitió hacer distintas aproximaciones hasta dar con la definitiva.

Alguna prueba, como la realización de un control por bloque, terminó por descartarse al no tener solución (aparentemente) sus problemas para redimensionar el tablero. Otras no permitían representar los patrones muy intuitivamente o sencillamente no tenían una apariencia que nos terminara de convencer.

Finalmente la opción elegida fue realizar un *control de usuario* que consiste en pintar el tablero con *GDI+*. La primera acción que realiza es detectar las dimensiones de su contenedor (la región de formulario en la que se situará). A partir de ellas, se calcula dinámicamente el tamaño que tendrán todas las líneas, y las posiciones de los bloques, casillas y números.

Al ser un control de este tipo, nos ha permitido ir añadiendo distintas representaciones de los patrones según los íbamos desarrollando con sólo variar su método `OnPaint`.

7.2.2. Problema de la consistencia de los candidatos

El segundo gran problema al que nos enfrentamos tenía que ver con la detección de los patrones de resolución.

Dado que uno de los principales objetivos del sistema es servir como tutor, debía ser capaz de interpretar el tablero que tiene el jugador y detectar un patrón sobre él. Como se vio en el capítulo 3, para detectar un patrón es necesario tener en cuenta los candidatos de cada casilla. Y aquí es donde aparece el problema:

> Es normal que el usuario tenga "errores" en los candidatos de cada casilla, bien porque se haya equivocado, o bien porque no los haya calculado todos todavía.

Sin embargo el sistema no puede calcularlos de nuevo cada vez que se le pida ayuda. Si lo hiciera, devolvería un tablero con más candidatos de los que tenía el usuario, o peor todavía, ofrecería como pistas deducciones que no son posibles con el tablero actual del jugador. Además, el cometido de la mayoría de los patrones es eliminar candidatos, y si se recalcularan en cada paso, se volverían a añadir.

Así pues, la solución fue que el sistema se inicializara con un tablero como el del jugador pero con todos los candidatos. Cada vez que se le pide ayuda se hace un proceso de actualización en el que se equiparan lo más posible, pero nunca se borra el candidato que será el valor final (aunque el usuario no lo tenga puesto).

Como hay ligeras diferencias entre el tablero del sistema y el del usuario, antes de devolver una pista o un paso se comprueba en el del usuario que efectivamente puede deducirla y además le servirá de algo aplicarla.

Un método adecuado de actualización de los candidatos, y estas comprobaciones adicionales de los patrones, ha dado como resultado un sistema capaz de ofrecer pistas coherentes o de explicar el motivo por el que no puede.

7.2.3. Patrones de resolución avanzados

Este problema radica en la complejidad de algunos patrones como los de *Coloring*, *XY-Chains* o sobre todo *Forcing Chains*, que presentamos en el capítulo 3. Los patrones de niveles inferiores de dificultad están perfectamente definidos y se puede encontrar documentación clara sobre ellos. Sólo queda entenderlos e implementarlos. Sin embargo estos patrones avanzados no están ni mucho menos estandarizados, cada cual los interpreta según le parece más lógico, y por tanto la labor de investigación se complica.

El caso más complicado fue el de las *Forcing Chains*. En los foros de programación de sudokus es una de las cuestiones más recurrentes, ya que es un patrón muy potente pero cuya complejidad algorítmica es demasiado elevada si se trata de hacer sin un método adecuado.

Como resultado de la búsqueda llegamos al trabajo de Eppstein ([6]) en el que se explican dos tipos de grafos y su aplicación a la resolución de sudokus. Detectar los patrones basados en cadenas con estos grafos también es costoso, pero lo mínimo que puede serlo, ya que se eliminan de la búsqueda todos los caminos que seguro que no servirá de nada explorar.

Tras hacer experimentos con estos dos tipos de grafos nos dimos cuenta de que por separado no eran tan potentes como esperábamos, y para obtener la potencia que queríamos había que combinarlos. Este proceso de combinación de ambos, con las restricciones que hay que imponer y las deducciones que se pueden obtener está explicado en el apartado 3.15.

7.2.4. Problema de la generación de tableros

El siguiente gran problema fue cuando tratamos de generar tableros clasificando por dificultad. Si de los patrones anteriores era complicado encontrar documentación al respecto, con este tema el problema se multiplicaba, ya que mucha menos gente lo ha conseguido, y los pocos que lo han hecho no revelan sus métodos.

Como se explica detalladamente en el capítulo 5, hay dos grandes métodos básicos para generar tableros:

- Uno consiste en rellenar tableros aleatoriamente partiendo del tablero vacío.
- El otro consiste en rellenar completamente un tablero e ir vaciando casillas aleatoriamente.

Además, en cada paso de rellenar casillas (o de vaciarlas en el segundo método) hay que resolver el tablero por el método más rápido posible para ver si tiene solución única. En nuestro caso, este método era el de la vuelta atrás.

Cualquiera de los dos métodos genera un tablero, pero para conseguir uno de una dificultad determinada, hay que repetir el proceso de creación hasta encontrarlo.

El proceso se hacía demasiado lento para hacer esperar a un usuario, por lo que hubo que investigar métodos más rápidos de resolución que el de la vuelta atrás. El más rápido conocido es la aplicación del método de los *Dancing Links*, que presentamos en el capítulo 5. Comprenderlo ya fue un problema en sí mismo, ya que utiliza unas estructuras de datos un tanto peculiares, pero su velocidad es impresionante y una aproximación basada en este método acabó aportando la solución.

7.3. Posibles ampliaciones

Gestión de usuarios

Una ampliación que podría ser interesante es la gestión de usuarios por parte de la aplicación. Ésta se encargaría de registrar en algún tipo de base de datos información estadística de cada usuario:

- Tableros resueltos y su nivel de dificultad.
- Tiempo medio y mínimo de resolución de tableros de cada nivel de dificultad.

Utilizando esta información se podrían realizar clasificaciones entre todos los usuarios, por niveles, tiempo necesitado, ...y representar gráficamente los resultados.

Generalización del tablero

Otro punto a tener en cuenta sería posibilitar la elección de las dimensiones del sudoku, es decir, no sólo limitar la aplicación a tableros de 9×9, sino poder elegir tableros de 16×16, y en general de $n^2 \times n^2$.

Los tableros ya están preparados para esta generalización, ya que incluyen una variable `NumCeldasPorFila` con la que se trabaja en las operaciones que se realizan sobre ellos.

El principal inconveniente se encuentra en la representación gráfica del sudoku. No conocer el número de casillas del tablero limitaría, o complicaría muchísimo, por ejemplo la representación de los patrones basados en cadenas, en los cuales se colorean y unen mediante rayas ciertos candidatos de algunas casillas.

Versión Web

Un proyecto interesante podría ser la realización de esta aplicación en versión web. El cambio se centraría principalmente en la capa de presentación pudiendo reutilizar el resto de las capas puesto que son independientes.

Dado que las aplicaciones web tienen una serie de limitaciones propias de la tecnología en la que se basan, para poder dotar al sistema de toda la funcionalidad e interacción con el usuario que tiene SuDoku JADE, habría que investigar tecnologías como *AJAX* (*Asynchronous JavaScript And XML*), que permite crear aplicaciones web interactivas.

7.4. Comparativa con otras aplicaciones

Debido a la gran variedad de programas relacionados con los sudokus, nos ha parecido interesante realizar un estudio comparativo entre algunas de estas aplicaciones y la que nosotros hemos desarrollado, con el fin de poder valorar, de cierta forma, el trabajo realizado.

Websudoku

Probablemente se trata de una de las aplicaciones más conocidas por los usuarios de internet. Se puede acceder a ella a través de la dirección web `http://www.websudoku.com`. Existen dos variantes: la versión web que es totalmente gratuita, y la versión normal que es de pago, aunque se puede probar durante una hora.

La versión web de esta aplicación está pensada para que los usuarios no dispongan de ayudas. Por ello, es capaz de generar tableros según cuatro niveles de dificultad, pero no es capaz de resolver tableros ni de guardarlos. La versión de pago sí es capaz de guardar, cargar y resolver tableros, pero no permite que el usuario introduzca tableros manualmente.

Ventajas Los puntos destacables de esta aplicación con respecto a otras son:

- Permite que el usuario introduzca candidatos en cada casilla.

- Dispone de un reloj para contar el tiempo que tarda el usuario en resolver un tablero particular.
- Permite volver al tablero inicial, en caso de que el usuario lo requiera.
- El usuario puede solicitar a la aplicación que compruebe si existe alguna casilla que viole alguna restricción del juego.

Todos estos puntos también se encuentran disponibles en nuestra aplicación.

Desventajas Hay otros aspectos que este programa no permite y que nosotros hemos considerado interesante incluir:

- No es capaz de ayudar al usuario a resolver un tablero mediante la generación de pistas o de resolución de pasos explicando la deducción lógica seguida.
- El uso de colores no está implementado.
- Permite la introducción de candidatos por parte del usuario, pero sólo cinco por casilla, cuando lo ideal sería que se pudieran introducir los nueve candidatos.
- No permite que el usuario pueda deshacer pasos.

Sudokusolver

Es una aplicación web que se centra en la resolución de tableros mediante el uso de patrones de resolución. También genera tableros con tres niveles de dificultad, y permite al usuario introducir sus propios tableros. Se puede acceder a ella en la dirección web `http://www.sudokusolver.co.uk/`.

La pega de esta aplicación es que los patrones implementados son suficientes para resolver tableros de niveles fácil, medio y algún tablero difícil, pero si le insertamos un tablero generado por nuestra aplicación de nivel superior al difícil, tiene que ayudarse de algún algoritmo de prueba y error para poder resolverlo.

Sadman's sudoku

Se trata de la aplicación más completa que hemos encontrado en la red. Es de pago pero permite su evaluación por un periodo de treinta días. Puede accederse a ella en `http://www.sadmansoftware.com/sudoku`.

Ventajas Los principales puntos fuertes de esta aplicación son los siguientes:

- Genera tableros con ocho niveles de dificultad.

- Es capaz de cargar y guardar tableros, y permite al usuario introducir manualmente uno.
- Implementa la mayoría de patrones de resolución que se conocen.
- Es capaz de resolver tableros utilizando patrones de resolución y algoritmos de prueba y error. Además, el usuario puede elegir qué patrones se pueden usar para resolverlos.
- Permite el uso de candidatos por parte del usuario, pudiéndose apuntar en cada casilla los nueve candidatos.
- Permite el coloreado de casillas por parte del usuario.
- Es capaz de dar pistas al usuario.
- Permite deshacer y rehacer acciones, y limpiar el tablero.

Leyendo estas características podría dar la impresión de estar leyendo la funcionalidad de nuestro programa SuDoku JADE. Ambas aplicaciones son muy similares, pero existen pequeñas diferencias entre ellas:

- Esta aplicación no implementa los patrones *XY-Wing*, *XYZ-Wing*, *XY-Chains*, y *Remote Pairs*.

- Genera pistas, pero no permite que el usuario elija el patrón con el que se buscan. Consideramos muy interesante permitir esto, ya que es una de las bases para aprender la aplicación de un patrón.

  Por ejemplo, si estamos resolviendo un tablero y creemos haber detectado un *Locked Candidates*, en nuestra aplicación podemos ver si efectivamente lo es seleccionándolo y generando pista. En esta no se podría hacer si hay un patrón más sencillo utilizable.

- A la hora de aplicar pistas con patrones que se basan en teoría de grafos, como el patrón *Forcing Chains*, nuestra aplicación dibuja líneas auxiliares que representan dichos grafos, presentando la pista al usuario de forma muy intuitiva.

  Sin embargo esta aplicación simplemente señala una casilla diciendo que sólo puede tener un valor, lo cual consideramos insuficiente para cadenas que no sean extremadamente sencillas.

- Nuestra función de resolver un tablero mediante patrones, muestra al usuario los patrones que se han requerido y el número de veces que se han utilizado. Con esta información el usuario puede hacerse una idea exacta del nivel de dificultad del tablero.

  Esta aplicación tiene una función de clasificar tablero que muestra al usuario el nivel del tablero, pero no muestra al usuario qué patrones utiliza a la hora de resolver tableros, lo cual no es tan descriptivo.

Sudoku Samurai de Koala Software

Esta aplicación comercial trata un tema un poco diferente, la variante samurai del sudoku, que consiste en cuatro tableros de sudoku unidos todos ellos por un tablero central. Aunque el problema que trata esta aplicación no es exactamente el mismo, vamos a realizar la comparativa debido a que es la única aplicación desarrollada por una empresa a la que hemos tenido acceso.

Este programa permite realizar las opciones básicas, como generar y resolver tableros, permitir al usuario introducir uno o guardar tableros en un fichero para poder cargarlos después. Sin embargo, no implementa otras funciones que consideramos básicas.

Los puntos más importantes de la comparación son los siguientes.

- No se permite al usuario anotar en cada casilla los candidatos que estime oportunos. Únicamente puede ver los que la aplicación calcula.
- Permite generar tableros con dos niveles de dificultad, fácil y difícil, mientras que nuestra aplicación permite generar tableros con seis niveles de dificultad.
- Realiza la resolución parcial del tablero a petición del usuario. Para ello utiliza los dos patrones de resolución más fáciles, mientras que nuestra aplicación hace uso de los patrones de resolución de todos los niveles de dificultad.
- No da pistas al usuario sobre el siguiente movimiento a realizar.
- No implementa el coloreado de casillas para ayudar al usuario a detectar patrones y resolver el tablero.
- La aplicación tiene una opción llamada *fijar tablero* que guarda la configuración de tablero actual en memoria, para poder volver a ella después. En cierto modo, es una manera de realizar la función deshacer. Sin embargo, nuestra aplicación guarda el tablero actual automáticamente, para poder volver a él después, mediante la función deshacer en caso de que el usuario lo crea conveniente.

Bibliografía

[1] Wikipedia, The Free Encyclopedia. *Sudoku*. 2006.
`http://es.wikipedia.org/wiki/Sudoku`

[2] Angus Johnson. *Solving sudoku*. 2005.
`http://angusj.com/sudoku/hints.php`

[3] SadMan Software. *Solving techniques*. 2006.
`http://www.sadmansoftware.com/sudoku/techniques.htm`

[4] Dan Rice's sudoku blog. *X-Wings*. 2005.
`http://sudokublog.typepad.com/sudokublog/2005/08/xwings.html`

[5] Dan Rice's sudoku blog. *The password is Swordfish!*. 2005.
`http://sudokublog.typepad.com/sudokublog/2005/08/the_password_is.html`

[6] David Eppstein. *Nonrepetitive Graphs and Cycles in Graphs with Application to Sudoku*. 2005.
`http://www.ics.uci.edu/~eppstein/pubs/graph-path.html`

[7] Donald E. Knuth. *Dancing Links*. 2000.
`http://www-cs-faculty.stanford.edu/~knuth/preprints.html`

[8] Wikipedia, The Free Encyclopedia. *Dancing Links*. 2006.
`http://en.wikipedia.org/wiki/Dancing_Links`

[9] Wikipedia, The Free Encyclopedia. *Algorithm X*. 2006.
`http://en.wikipedia.org/wiki/Algorithm_X`

[10] DLX Engine. *Re-use of Existing Java Code (Sudoku Engine)*. 2006.
`http://blogs.msdn.com/dotnetinterop/archive/2006/02/15/532103.aspx`

[11] Craig Larman. *Applying UML and Patterns*. Prentice-Hall, 2001.

[12] Erich Gamma et al. *Design Patterns*. Addison-Wesley Professional, 1995.

[13] Andrew Troelsen. *Pro C# 2005 and the .NET 2.0 Platform*. Appress, 3rd edition, 2005.

[14] Mahesh Chand. *GDI+ programming with C#*. Addison-Wesley, 2003.

[15] Chris Sells. *Windows Forms programming in C#*. Addison-Wesley, 2003.

[16] The Code Project. *Links with arbitrary text in a RichTextBox*. 2005. `http://www.codeproject.com/cs/miscctrl/RichTextBoxLinks.asp`