UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS

SudokuSolver

Análisis y Diseño de Algoritmos I
Trabajo Final

Orlando, José Ignacio
(E-mail: nacho_glorfindel@hotmail.com)
Rodríguez, Ana Victoria
(E-mail: mimi244@gmail.com.ar)

Docentes a cargo:
Claudia Pereira
Liliana Martínez

27 de febrero de 2009

Índice

1. Introducción

El objetivo del presente trabajo final es el desarrollo de un programa en C++ que resuelva el tradicional juego del Sudoku a través de la técnica backtracking, y que permita, además, que el usuario pueda encontrar por sus propios medios la solución. Para el tratamiento del mismo se tuvieron en cuenta los conocimientos adquiridos en materia de análisis, diseño e implementación de tipos de datos abstractos (TDA), de eficiencia de algoritmos, de la estrategia aplicada y de estructuras de datos. La aplicación final trabaja en un entorno visual elegante que facilita su uso respecto de otras alternativas. Además, se incorporó una herramienta adicional para la construcción de tableros Sudoku, de forma tal que éstos puedan probarse sin ninguna clase de inconvenientes.

El presente informe tiene como objetivo acompañar al lector en el estudio del comportamiento del programa, dotándolo de las herramientas necesarias para analizar el problema en términos aritméticos y algorítmicos, como así también acompañarlo durante el análisis con comentarios sobre el código final y su ejecución.

Hemos dividido esta monografía en cuatro secciones básicas. En el apartado de "Análisis" se otorgan al lector las herramientas necesarias para adquirir toda la información correspondiente al juego del Sudoku, desde datos históricos hasta análisis particulares enfocados a sus diversas áreas de estudio, haciéndose una referencia detallada de las estrategias de resolución del juego que más se emplean, de acuerdo a la información recopilada. En "Diseño" se abordan cuestiones tales como el diseño de clases y tipos de datos abstractos, su especificación en lenguaje Nereus, algunas alternativas de implementación y un estudio abstracto de los algoritmos de backtracking que resuelven el juego, analizando también la complejidad temporal en términos de notación asintótica. En "Implementación" se hace referencia al desarrollo final del programa, así como también se detallan aspectos de importancia sobre el código fuente. Por último, en "Conclusiones" se aborda un estudio mucho más completo de la eficiencia de los algoritmos, acompañado de las conclusiones finales a las que llegamos al término del desarrollo de este trabajo. Se incorpora además una sección extra, "Referencias", que indica la bibliografía consultada para realizar el trabajo.

Aclaramos, además, que se utilizó para el desarrollo del algoritmo el compilador de wxDev-C++, complemento del tradicional Bloodshed Dev-C++ que facilita la creación de aplicaciones gráficas, y cuyo instalador se incluye en el CD del trabajo.

2. Análisis

2.1. Historia

El Sudoku puede haberse ideado a partir de los trabajos del matemático suizo Leonhard Euler (1707-1783), quién utilizó los cuadrados latinos[1] para realizar cálculos probabilísticos, aunque su aparición más obvia se dio en el año 1979, cuando el norteamericano Howard Garns publica en Nueva York, a través de la empresa Dell Magazines, un juego muy similar conocido bajo el nombre de *"Number Place"*. En el año 1984 una editorial japonesa lo exportó hacia ese país, divulgándolo en el periódico *"Monthly Nikolist"* bajo el nombre de *"Los números deben estar solos"* (cuyo apócope en japonés es *SuDoku*). La editorial introdujo con el correr de los años algunas innovaciones que hicieron que el juego ganara gran popularidad entre los lectores. [1]

Sus inicios reales como Sudoku en Occidente se remontan a los años 2004/2005, gracias a su publicación en los diarios "*The Times"* y "*The Daily Mail"*. Desde aquel entonces, el juego ha alcanzado una gran popularidad en el mundo moderno: en 2005 se publicó el primer libro acerca de Sudokus en el mundo (*"Los mejores Sudokus"*, con 200 tableros agrupados en 4 niveles de dificultad) y se lo incluyó entre los 9 problemas del ICPC (sigla de *International Collegiate Programming Contest*). [1]

En nuestro país, el diario *"Clarín"* publica Sudokus en sus juegos de contratapa, junto con la tradicional *"Claringrilla"* y los autodefinidos.

2.2. Introducción al juego

El Sudoku es un juego que consiste en un tablero de $R \times C$ casillas, donde R nota el número de filas y C el número de columnas. Dicho tablero está dividido en k subtableros o bloques de tamaño $r \times c$ (con $r = \sqrt{R}$ y $c = \sqrt{C}$) en los que hay que ubicar $r \times c$ elementos diferentes de acuerdo a los postulados de una regla que se conoce con el nombre de *"the One Rule"* [2], o "Regla Única". Dicha regla asevera que *cada fila, columna o bloque del sudoku debe contener una y solo una vez cada elemento indistinguible*. Si llamamos N al conjunto de estos elementos (por lo general, la sucesión de números naturales $\{1, 2, \dots, n\}$) podemos expresar la regla en términos matemáticos:

$$\forall n \epsilon N : \; n\epsilon(i,j) \Leftrightarrow m \neq n, \forall m\epsilon(i,h) \vee \; \forall m\epsilon(k,j) \vee \; \forall m\epsilon Bloque(i,j)$$

$$0 \leq i,k \leq r, \qquad 0 \leq h,j \leq c$$

[1] Los cuadrados latinos son los primeros tableros similares a los del Sudoku de los que se tiene referencia histórica. Para más información consultar el libro *"On magic squares"*, de Leohnard Euler [5].

La versión más popular del sudoku consta de un tablero de $3^2 \times 3^2$casillas (un total de 81), dividido en bloques cuadrados de tamaño 3×3, y donde el conjunto de elementos indistinguibles es nada más y nada menos que $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$. En la *figura 1* se muestra un tablero sudoku clásico y vacío, con sus distintas partes correctamente referenciadas.

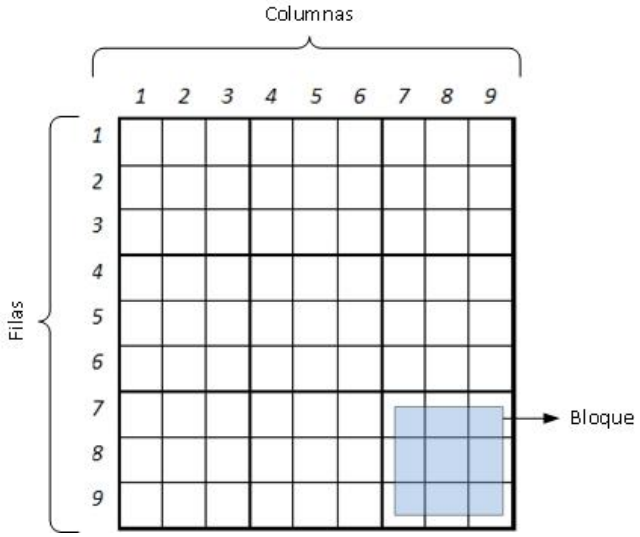

Figura 1: *Tablero sudoku de 9 x 9 casillas, vacío, con sus distintas regiones identificadas.*

Existen otras versiones de sudokus en las cuales aumenta el número de restricciones para las soluciones, se introducen variantes respecto de las regiones o incluso cambia el conjunto de elementos a utilizar (por ejemplo, el *Sudoku nonomino* tiene los bloques con formas irregulares que se distinguen entre sí a través de colores) [1].

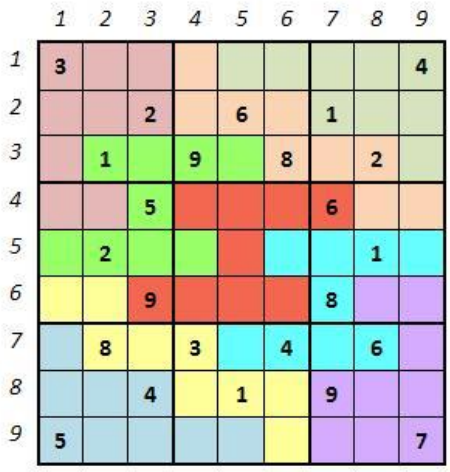

Figura 2: *Un ejemplo de tablero Sudoku nonomino. Se observa con facilidad la diferencia entre los bloques comunes y los de color.*

Otro detalle a tener en cuenta es que los tableros sudoku vienen precargados con un conjunto de valores numéricos que reciben el nombre de "*clues*" (pistas) [2]. Éstas no se caracterizan por haber sido colocadas en un orden aleatorio puesto que no cualquier tablero de 81 casillas con números dispersos puede ser un sudoku, a sabiendas de que *un tablero sudoku es válido sí y sólo sí admite una y solo una solución*. Luego, la construcción de un tablero sudoku implica, en la mayoría de los casos, que al término de la misma se conozca la solución que lo satisface. En la *figura 3* se observa el sudoku conocido como "Star Burst Leo", formado por 20 pistas.

Por lo general, los sudokus se clasifican según un determinado nivel de dificultad. Es importante considerar, sin embargo, que la dificultad de un tablero dado no depende exclusivamente de la cantidad de casillas-pista que éste posee, sino también del tiempo que toma encontrar la solución [6]. Cuando se quiere determinar la complejidad de un tablero en términos del tiempo que llevaría a un ser humano resolverlo, suele solucionárselo utilizando las reglas que se enuncian en la sección 2.4.2, contando la cantidad de pasos necesarios y realizando un muestreo estadístico por intervalos del número de iteraciones de cada tablero. Luego, bastará con darle un nombre a cada intervalo y clasificar los tableros según éstos.

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

Figura 3: *"Star Burst Leo", un tablero sudoku con 20 casillas-pistas iniciales.*

2.3. Estudio matemático

Matemáticamente se han demostrado un conjunto de propiedades de los tableros de sudoku [4]. Entre ellas se destacan:

- El intercambio de filas o columnas de un mismo bloque de un Sudoku válido mantiene la condición de válido del nuevo tablero y la solución se obtiene con los mismos cambios sobre la solución original.

- Si se intercambian tres filas (columnas) que pertenecen a un mismo bloque por otras filas (columnas) que cumplen las mismas condiciones, se mantiene la validez del tablero y la solución se obtiene aplicando las mismas operaciones.
- La solución de un sudoku traspuesto es igual al traspuesto de la solución del sudoku original.
- Si un sudoku tiene una solución única, al agregarle más elementos de esa misma solución, ésta permanece única, y su dificultad se mantiene o disminuye, pero no crece.

Los tableros sudoku, como se ha dicho antes, están formados por un conjunto de casilleros o celdas vacías y un conjunto de celdas precargadas o pistas, formadas por algún número de la colección de números a ubicar. La cantidad de tableros de 9×9 casillas que pueden obtenerse ha sido calculada por los matemáticos Felgenhauer y Jarvis y asciende a unos $6.670.903.752.021.072.936.960 \approx 6{,}7 \times 10^{21}$ [2][2]. Para obtener dicho resultado se utilizaron complejas ecuaciones que involucran números combinatorios, sumatorias y factoriales cuyo análisis no tiene mucho sentido en este informe. No obstante, cabe mencionar que se ha desarrollado matemáticamente un algoritmo para obtener estos tableros sudoku de forma aleatoria, a través de las conocidas tablas de Cayley para grupos finitos (utilizadas en Álgebra I para el cálculo de sumas/productos de grupos $\mathbb{Z}_n$), lo cual demuestra que este problema tiene una solución conocida [2].

Otro de los estudios matemáticos que se realizan sobre los Sudokus está relacionado con la mínima cantidad de casillas-pista que debe tener un tablero para que admita una única solución. Según investigaciones basadas en algoritmos de fuerza bruta, los Sudokus ordinarios de 9×9 casillas soportan un mínimo de 17 pistas; de todas formas el análisis continúa abierto, pues se supone que existan sudokus de 16 pistas con solución única, o aún menores. Cuando el número de restricciones sobre la solución aumenta (otros tipos de sudokus), la cantidad mínima de casillas pistas necesarias para resolverlo merma [2].

2.4. Estudio algorítmico

El Sudoku tiene asociados dos problemas básicos: la generación de tableros válidos y la búsqueda de una solución que los satisfaga.

Respecto de la generación de tableros, puede decirse que existen dos alternativas intuitivas para darle respuesta:

- Mantener almacenados en una base de datos un conjunto de tableros que se conocen de antemano como válidos, y realizar una carga aleatoria de los mismos. La principal ventaja de esta estrategia es que el costo computacional se reduce exclusivamente al de generar un número aleatorio y realizar la carga del tablero correspondiente, aunque se produce un retardo a causa del tiempo que toma la lectura de un fichero

[2] En [3] hay disponible un algoritmo en ActionScript 2 que permite obtener alrededor de 16.000 billones de tableros Sudoku válidos a través de un método que se analizará en la sección 2.4.

del disco. La desventaja está en que el programa final ocupa un espacio extra en disco, supeditado al formato y codificación de los archivos.

- Desarrollar un algoritmo capaz de generar los tableros por sí mismo.

La solución ideal involucra una combinación de ambas estrategias [3]: a partir de una base de datos de tamaño acotado, formada por un conjunto de tableros cuya validez es conocida, y considerando las propiedades desarrolladas en detalle en la sección 2.3, se realizan operaciones aleatorias de permutaciones válidas de filas y columnas y trasposiciones de la matriz-tablero. Luego, el tablero resultante será claramente válido, puesto que su solución podrá obtenerse a partir de la solución del tablero inicial, aplicando sobre ésta las mismas operaciones que se utilizaron con la matriz. El costo computacional de este algoritmo es polinomial, y depende exclusivamente del tamaño del tablero y de la cantidad de operaciones que se quieran realizar. La implementación de este algoritmo está disponible en [3]. Nuestro trabajo utiliza para resolver este problema una base de datos de más de cien sudokus, algunos de ellos célebremente estudiados, para poder analizar la eficiencia de los algoritmos finales.

Respecto de la búsqueda de una solución, está demostrado que se trata de un problema NP-completo [7], con lo cual no se ha descubierto hasta el día de hoy un algoritmo que le de solución en un tiempo polinomial. Sin embargo, el Sudoku se ha convertido en uno de los principales problemas de estudio de los programadores, quienes continúan desarrollando formas de resolución cada vez más eficientes. A continuación explicaremos los más conocidos, abocándonos en profundidad al que involucra la técnica backtracking, que es el que utilizamos en nuestro trabajo.

2.4.1. Técnica backtracking

El algoritmo implementado con la técnica backtracking (en realidad, con *"Branch-&-Bound"* o ramificación y poda[3]) es uno de los más elogiados por los programadores en Internet por la velocidad con la que encuentra la respuesta (gracias al hardware con el que se dispone actualmente). Sin embargo, para los investigadores se trata del menos útil a causa de su proverbial ineficiencia.

Una de las alternativas que existen utilizando la técnica backtracking es la de emplearlo en su forma más primitiva, que es la de un algoritmo de fuerza bruta. En este caso, el algoritmo se encargará de probar en cada casilla, y a partir de una casilla inicial, cada uno de los valores disponibles, analizando inmediatamente la validez de la decisión tomada. Si la solución parcial no satisface las condiciones de la Regla Única, el algoritmo retrocederá (*backtrack*) al nivel de búsqueda anterior y generará otro tablero, con el siguiente valor disponible ubicado en la posición última de análisis. Este proceso se repetirá hasta concretar el análisis de la última casilla (para nuestro tablero, la número 81), con lo cual el costo computacional será de $n^{(R\times C)-p}$, con n igual al número de valores posibles, R el número de filas, C el número de columnas y p el número de casillas con pista. A este costo se debe agregar, también, el de cada consulta de validez (esto es, el costo de verificar si la casilla pertenece a una fila, a una

[3] La técnica de Branch-&-Bound se basa en una búsqueda exhaustiva como la del backtracking, solo que introduce criterios de poda para reducir el espacio de soluciones posibles.

columna o a un bloque), con lo cual el número es extremadamente grande. Para determinados tableros, entonces, el algoritmo tardará mucho tiempo, aún con el hardware actual. La ventaja, sin embargo, radica en que el algoritmo devolverá, tarde o temprano, la solución esperada. En la *figura 4* mostramos el espacio de soluciones (parcial) para un pequeño sudoku de ejemplo de $2^2 \times 2^2$ casillas, con $N = \{1, 2, 3, 4\}$.

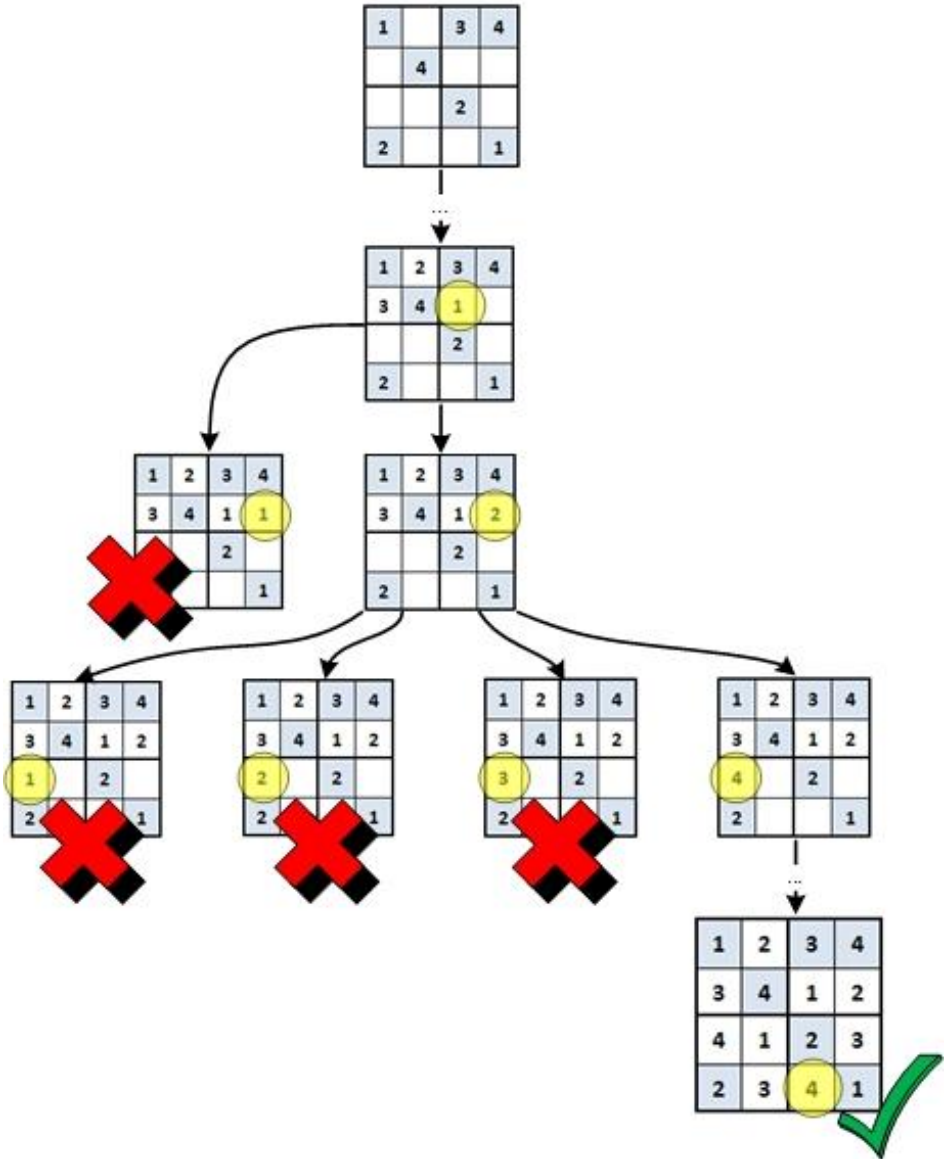

Figura 4: *Espacio de soluciones (acotado) para un tablero sudoku de 4 x 4 de ejemplo. Obsérvese que el algoritmo de backtracking no expande los nodos que no son válidos, y corta la búsqueda al encontrar la solución.*

Otro de los algoritmos basados en backtracking consiste en relacionar la búsqueda de una solución a un tablero sudoku con el clásico problema de coloreo de grafos [2] [8]. Un grafo $G = \langle V, E \rangle$ es una estructura formada por un conjunto de vértices V y un conjunto de aristas o arcos E que los unen. El problema de coloreo de grafos es un problema NP, y consiste en hallar la cantidad mínima de colores necesarios para colorear cada vértice del grafo, teniendo en

cuenta que dos vértices adyacentes (unidos entre sí por un arco) no pueden ser coloreados de la misma forma.

Si tenemos en cuenta que el conjunto de colores será el de los valores que se utilizan para completar el sudoku, el *quid* de esta relación radica en hallar una forma para representar el tablero sudoku con un grafo tal que cumpla las condiciones del problema de coloreo.

Sea V el conjunto de vértices de nuestro grafo. Si consideramos $V = \{(i,j),\ 0 \leq i,j \leq 9\}$ (es decir, tomamos cada casilla del tablero como un vértice del grafo), podemos construir E (el conjunto de aristas o arcos) de acuerdo al siguiente criterio: *"dos vértices u y v están unidos por un arco no dirigido (u,v) sí y solo sí u y v comparten la misma fila, columna o bloque".* En la *figura 5* se observa, para el sudoku de ejemplo utilizado anteriormente, cómo queda conformado el subgrafo de vértices adyacentes para la casilla $(1,1)$ siguiendo el criterio enunciado.

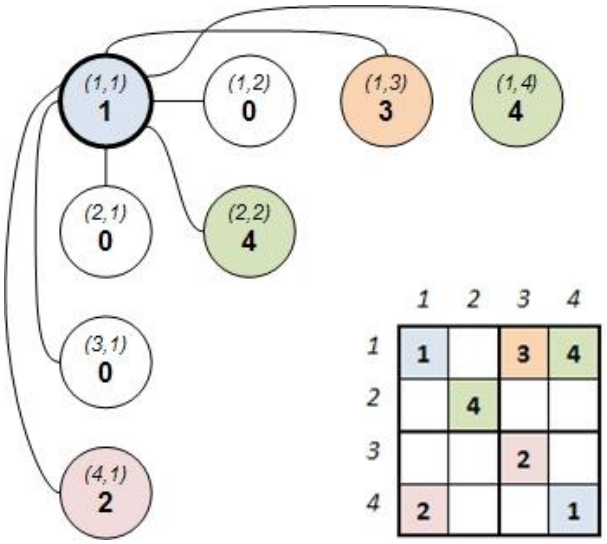

Figura 5: *Subgrafo creado a partir de las casillas relacionadas con la (1,1). Se omitieron los vínculos entre todos los demás vértices por cuestiones de prolijidad.*

Luego, el algoritmo de backtracking debe solo recorrer el grafo G obtenido coloreando de forma válida cada uno de los vértices, a partir del conjunto N de "colores" o de cifras del sudoku. En nuestro caso no tendremos que encontrar el conjunto mínimo de colores, sino buscar un coloreo válido para el grafo. Como todo tablero sudoku debe admitir una y sólo una solución (como vimos en la sección 2.3), queda asegurado que si el tablero es válido, el algoritmo devolverá un resultado satisfactorio en un tiempo exponencial, cuya expresión es una función de la cantidad de vértices que restan colorear (cantidad de casillas vacías del sudoku) y de la cantidad de colores disponibles (cantidad de valores con los que puede completarse una casilla del sudoku).

2.4.2. Método humano

El método humano de resolución es muy interesante para utilizarse algorítmicamente, debido que permite deducir cómo el usuario podría resolver un sudoku por su cuenta. Andries E. Brower publicó en la revista *"NAW"* [9] siete reglas básicas para el desarrollo del análisis humano. Entre ellas, las más simples (y más utilizadas) son: "*baby steps"* (cuando se conocen ocho cifras de una misma fila, columna o bloque, el dígito restante es el que ocupa la última casilla libre), *"singles"* (cuando hay un solo lugar para un elemento dado en una columna, fila o bloque dada/o, o cuando hay solo un dígito que puede ir en un bloque dado, escribirlo allí) y *"pair markup"* (si en una casilla pueden ir dos o más valores posibles, escribirlos como subíndices de forma tal que puedan realizarse los análisis correspondientes en el resto del sudoku: si se da con una contradicción al considerar uno de los dígitos, entonces corresponderá detener el análisis de ese y probar con alguno de los restantes). Está demostrado que siguiendo estos tres pasos (más otros cuatro de una complejidad algo mayor) puede encontrarse la solución a cualquier sudoku. La dificultad que presenta este algoritmo es que no puede calcularse a ciencia cierta su complejidad temporal, puesto que los pasos deben ejecutarse en un orden lineal y, cuando se usa un método de nivel inferior, deben repetirse luego todos los anteriores. Así, para algunos tableros, la solución se alcanzará en tiempos mínimos, mientras que para otros el análisis puede tornarse mucho más largo. Por último, podemos afirmar que la eficiencia del algoritmo dependerá también de la estrategia utilizada para la codificación de los pasos, puesto que serán sus componentes principales. El método humano se utiliza para calcular la dificultad de los sudokus que se publican en libros y revistas para su resolución "a mano".

2.4.3. Búsqueda aleatoria

La búsqueda aleatoria es otro de los métodos que pueden seguirse para encontrar el resultado de un sudoku. La estrategia consiste en trabajar de la siguiente manera: primero, comenzar aleatoriamente colocando números en las casillas vacías del tablero, y calcular luego el número de errores cometidos. Una vez terminado este proceso, se "intercambian" los números insertados a lo largo de la grilla hasta que la cantidad de errores se reduzca a cero. Cuando se llegue a esto, se habrá dado con una solución. La ventaja de este método, al igual que la del backtracking, es que el tablero no tiene que ser "soluble lógicamente" para que el algoritmo pueda resolverlo. Esto es, a diferencia de otras estrategias, los tableros con los que se provee al algoritmo no deben ser especialmente construidos de manera tal que le provean suficientes pistas al usuario como para completarlo usando solamente la lógica encadenada explicada en la sección 2.4.2. Es el menos utilizado, puesto que se trata de una búsqueda más bien azarosa y para nada sistemática.

2.4.4. Exact Cover

Así como explicamos anteriormente que un sudoku puede entenderse como una instancia del problema de coloreo de grafos, el sudoku puede describirse también en términos de un caso particular del problema de *"Exact Cover"*, o recubrimiento exacto. [8]

"Exact Cover" es un problema NP-completo que consiste en, dado un universo de elementos U y una colección S de subconjuntos de U, hallar una subcolección $S^* \subseteq S$ tal que cada elemento de S esté exactamente en un conjunto $s \subseteq S^*$. Por tratarse de un problema que no se resuelve en un tiempo polinomial, pueden plantearse soluciones por aproximación o definirse un algoritmo de backtracking que le de respuesta. Como alternativa a estas dos opciones, el Dr. Donald Knuth, cuya bibliografía ha sido de gran utilidad durante las cursadas de Análisis y Diseño de Algoritmos I y II, publicó hace ya algunos años un algoritmo que recibe el nombre de *"Dancing Links"* (algo así como "vínculos danzantes" o "vínculos móviles"), que resuelve el problema de *"Exact Cover"*, obviamente en un tiempo exponencial. En internet hay innumerables sitios que se encargan de implementar este algoritmo para la resolución de sudokus [14].

La relación entre la resolución de sudokus y el problema de *"Exact Cover"* se basa en definiciones de conjuntos y en términos matemáticos, y no se desarrollará aquí por cuestiones de espacio. Sin embargo, una excelente explicación matemática del problema está disponible en [8].

3. Diseño

3.1. Tipos de datos abstractos

3.1.1. Introducción a los tipos de datos abstractos

Durante el análisis del problema en cuestión se reconocieron distintos TDAs cuyo comportamiento se asemeja al de otros tipos de datos utilizados con frecuencia (arreglos, matrices, listas, pares, etc.). A los efectos de hacer más comprensible la implementación del trabajo y reducir incrementos innecesarios de los tiempos constantes, se utilizaron librerías tales como la STL (*Standard Template Library*), cuyos métodos se caracterizan por una gran eficiencia. Para los casos en los cuales existían diferencias o las estructuras presentaban una mayor complejidad respecto de las originales, se les incorporaron las funcionalidades extra. De esta forma, el análisis que realizaremos a continuación es meramente teórico, y puede no verse reflejado de manera directa en el código fuente.

3.1.2. Funcionalidades de los TDAs

Los tipos de datos que se reconocieron a la hora de diseñar el trabajo fueron los siguientes:

I) ConjuntoCasillas
Es una estructura que posee 81 casillas a las que se identifican unívocamente, y que pueden ser o no pistas. Estas casillas se encuentran asociadas a través de su identificador en filas, columnas y bloques que se corresponden con el sudoku tradicional implementado. Además, posee todos los métodos necesarios para poder resolver el juego propuesto. Sus funcionalidades son:

- *ConjuntoCasillas():*
  Método constructor de la clase que crea el conjunto de 81 casillas identificadas y asociadas, siendo todas incógnitas y seteadas con el valor "?".
- *void SetCasillaNueva(IdentificadorCasilla Id, bool Es_Pista, int Valor)*
  Método modificador que asigna el entero Valor a una casilla identificada con Id y un booleano determinando si se trata o no de una casilla pista.
- *void SetCasillaIncognita(IdentificadorCasilla Id, int Valor)*
  Método modificador que asigna el entero Valor a una casilla conocida de antemano como incógnita (es decir, no-pista), e identificada con un identificador de casilla. En la representación más trivial del conjunto de casillas, un identificador de casilla es básicamente un par (x, y), con $1 \leq x, y \leq 9$.
- *int GetValorCasilla(IdentificadorCasilla Id)*
  Método observador que, dada una casilla identificada con Id, retorna el valor entero que ha sido colocado previamente en ella.
- *bool GetPistaCasilla(IdentificadorCasilla Id)*
  Método observador que retorna un booleano indicando si la casilla identificada a través de Id es o no una casilla pista.

- *unsigned int BackResolucion(unsigned int & Iteraciones, unsigned int & Cantidad_Soluciones)*
  Método modificador que resuelve el tablero sudoku. Obsérvese que en Iteraciones se devolverá el número de ciclos ejecutados para encontrar la solución, y que en cantidad de soluciones se determinará el número de soluciones buscadas. El algoritmo, además de modificar el conjunto de casillas de forma tal que permanezca contenida la solución obtenida, retornará un entero indicando la cantidad de soluciones que no pudieron hallarse. Para analizar más en detalle las necesidades de esta funcionalidad, léase con detenimiento la sección 4.
- *bool Verificar(IdentificadorCasilla & Casilla1, IdentificadorCasilla & Casilla2)*
  Método observador que devuelve un booleano indicando si el tablero es o no una solución. Si detecta errores tales como repetición de valores en fila, columna o bloque o casillas que aún son incógnitas, devolverá los identificadores de las casillas erróneas en Casilla1 y Casilla2.
- *bool ValidacionDePistas()*
  Método observador que verifica que las pistas dadas no se repitan en una misma fila, columna o bloque.

II) Conjunto de tableros:
La clase se comporta como una colección de tableros sudoku, permitiéndose así tener agrupados en un único sitio a varios de ellos, clasificados de acuerdo a un identificador que cada uno posee. Sus funcionalidades principales son:

- *ConjuntoTableros():*
  Constructora básica. Crea un entorno vacío para la colocación posterior de tableros.
- *void SetTablero(Tablero nuevoTablero):*
  Método constructor que permite colocar en el conjunto un tablero dado.
- *Tablero GetTablero(IdentificadorTablero Id):*
  Método observador que permite, dado un identificador de tableros, obtener el tablero de la colección cuyo identificador coincide. Como nuestro algoritmo para la apertura de tablero tiene dos opciones (apertura y elección al azar), resulta conveniente representar al identificador de los tableros como un número entero que sea generado por el código del algoritmo.

III) Tablero:
Es una estructura formada por un nombre, un cierto nivel de dificultad (para su resolución con el método humano) y un conjunto de casillas que contienen valores incógnita (celdas que deben ser llenadas por el usuario o por el algoritmo de resolución) o pistas (que no pueden ser modificadas). Sus principales funcionalidades son:

- *Tablero():*
  Constructora básica que crea el entorno necesario para comenzar a armar la estructura.
- *void SetNombre(string Nombre)*
  Método modificador que, dado un string, lo asigna al tablero como su nombre.
- *void SetDificultad(string Dificultad)*

Método modificador que, dado un string, lo asigna al tablero como su dificultad en ser resuelto utilizando las 7 reglas básicas de la lógica encadenada.

- *void SetCasillaIncognita(IdentificadorCasilla Id, int Valor)*
  Método modificador que llama internamente al método ConjuntoCasillas:: SetCasillaIncognita().
- *void SetCasilla(IdentificadorCasilla Id, int Valor, bool EsPista)*
  Método modificador que invoca internamente al método ConjuntoCasillas:: SetCasillaNueva().
- *int GetCasilla(IdentificadorCasilla Id)*
  Método observador que invoca internamente al método ConjuntoCasillas:: GetValorCasilla().
- *bool GetEsPista(IdentificadorCasilla Id)*
  Método observador que invoca internamente al método ConjuntoCasillas:: GetPistaCasilla().
- *string GetNombre():*
  Método observador que devuelve el nombre del tablero.
- *string GetDificultad():*
  Método observador que devuelve la dificultad del tablero.
- *unsigned int Resolver (unsigned int & Iteraciones, unsigned int & cant_soluciones)*
  Método modificador que invoca internamente al método ConjuntoCasillas:: BackResolucion().
- *bool Verificar (IdentificadorCasilla & Casilla1, IdentificadorCasilla & Casilla2)*
  Método observador que invoca internamente al método ConjuntoCasillas::Verificar().
- *bool PistasValidas()*
  Método observador que invoca internamente al método ConjuntoCasillas::ValidacionPistas().

IV) Escenario

Representa el espacio del juego en cualquier instante del programa, y es a su vez la interfaz que existe entre el usuario del juego y el tablero en sí mismo. Sus métodos son:

- *Escenario():*
  Constructora básica. Genera un escenario vacío que posteriormente se completará con un tablero y con las modificaciones que el usuario o el algoritmo realicen sobre éste.
- *void SetTablero(Tablero nuevoTablero):*
  Método constructor que, dado un tablero, lo presenta en el escenario de juego para su posterior resolución.
- *void SetValorEnCasilla(IdentificadorCasilla Id, int Valor):*
  Método modificador que invoca internamente al método Tablero::SetCasillaIncognita. Obsérvese que se utilizará para que el usuario introduzca valores dentro del tablero durante el proceso de resolución manual.
- *void Resolver()*
  Método modificador que resuelve el tablero activo. Obsérvese que invoca internamente al método Tablero::Resolver(), para luego actualizar la información en

pantalla con los resultados obtenidos. En el parámetro cant_soluciones se pasará un 1, teniendo en cuenta que se quiere dar con la única solución del tablero.

- *bool Verificar()*
  Método observador que invoca internamente al método Tablero::Verificar().
- *void Deshacer()*
  Método modificador que deshace el último cambio hecho en el ConjuntoCasillas.

V) Editor

Se trata de una interfaz que le permite al usuario crear sus propios tableros. Son funciones del editor:

- *Editor():*
  Constructora básica. Genera un espacio formado por un conjunto de casillas vacías y reserva un espacio para poder colocar un nombre y un nivel de dificultad.
- *void SetValorEnCasilla(IdentificadorCasilla identif, int valor):*
  Método modificador que invoca internamente al método Tablero::SetCasillaNueva(). Obsérvese que se utilizará para que el usuario introduzca valores dentro del tablero durante el proceso de armado del tablero.
- *void SetNombre(string nombre):*
  Método modificador que, dado un string, lo setea como nombre del posible tablero.
- *void SetDificultad(string dificultad):*
  Método modificador que, dado un entero, lo setea como dificultad del posible tablero.
- *bool Guardar():*
  Método modificador que setea inicialmente las casillas con número como pistas y que luego verifica que el número de pistas ingresadas sea mayor a 16, que éstas no generen contradicción (utilizando Tablero::PistasValidas()) y, a través del método del Tablero::Resolver(), con cantidad_soluciones = 2, si el tablero propuesto tiene una única solución. De ser así, lo guarda en un archivo de texto y retorna true, reseteando en "?" todas las casillas del conjunto y volviéndolas no-pista para poder continuar creando tableros. Caso contrario, devuelve false y vuelve las casillas a su estado de no-pista para ser modificadas.

3.1.3. Especificaciones en lenguaje Nereus

El lenguaje Nereus se utiliza en el diseño de sistemas para realizar especificaciones algebraicas de tipos de datos abstractos. Así, a través de una estructura y una codificación normalizadas, pueden leerse con claridad los métodos y atributos de las clases involucradas. Utilizando este lenguaje, especificamos a continuación los tipos de datos enunciados en la sección 3.1.2.:

CLASS ConjuntoCasillas
IMPORTS Integer, Boolean, Par
BASIC CONSTRUCTORS ConjuntoCasillas
EFFECTIVE
TYPES ConjuntoCasillas
FUNCTIONS
ConjuntoCasillas: → ConjuntoCasillas
SetCasillaNueva: Par x Boolean x Integer x ConjuntoCasillas → ConjuntoCasillas
SetCasillaIncognita: Par x Integer x ConjuntoCasillas → ConjuntoCasillas

GetValorCasilla: Par x ConjuntoCasillas → ConjuntoCasillas
GetPistaCasilla: Par x ConjuntoCasillas → ConjuntoCasillas x Boolean
BackResolucion: Integer x Integer x ConjuntoCasillas → ConjuntoCasillas x Integer x Integer
Verificar: Par x Par x ConjuntoCasillas → Boolean
ValidacionDePistas: ConjuntoCasillas → Boolean

AXIOMS

...

END-CLASS

CLASS ConjuntoTableros
IMPORTS Tablero, IdentificadorTablero
BASIC CONSTRUCTORS ConjuntoTableros
EFFECTIVE
TYPES ConjuntoTableros
FUNCTIONS

ConjuntoTableros: → ConjuntoTableros
SetTablero: Tablero → ConjuntoTableros
GetTablero: ConjuntoTableros x IdentificadorTablero → Tablero

AXIOMS

...

END-CLASS

CLASS Tablero
IMPORTS String, Integer, Boolean, ConjuntoCasillas
BASIC CONSTRUCTORS Tablero
EFFECTIVE
TYPES Tablero
FUNCTIONS

Tablero: → Tablero
SetNombre: String x Tablero → Tablero
SetDificultad: String x Tablero → Tablero
SetCasillaIncognita: Par x Integer x Tablero → Tablero
SetCasilla: Par x Integer x Boolean x Tablero → Tablero
GetCasilla: Par x Tablero → Integer
GetNombre: Par x Tablero → String
GetDificultad: Par x Tablero → String
Resolver: Integer x Ineteger x Tablero → Integer x Tablero
Verificar: Par x Par x Tablero → Boolean
PistasValidas: ConjuntoCasillas → Boolean

AXIOMS

...

END-CLASS

CLASS Escenario
IMPORTS Tablero, Par, Integer, Boolean
BASIC CONSTRUCTORS Escenario
EFFECTIVE
TYPES Escenario
FUNCTIONS

Escenario: → Escenario
SetTablero: Escenario x Tablero → Escenario
SetValorEnCasilla: Par x Integer x Escenario → Escenario
Resolver: Escenario → Escenario
Verificar: Escenario → Boolean
Deshacer: Escenario → Escenario

AXIOMS

...

END-CLASS

CLASS Editor
IMPORTS Tablero, Par, Integer, Boolean
BASIC CONSTRUCTORS Editor
EFFECTIVE
TYPES Editor
FUNCTIONS

Editor: → Editor
SetNombre: String x Editor → Editor
SetDificultad: String x Editor → Editor

SetValorEnCasilla: Par x Integer x Editor → Editor
Guardar: Editor → Editor

AXIOMS

...

END-CLASS

3.1.4. Diagrama de relaciones de los tipos de datos

En la *figura 6* se observa un diagrama de relaciones con formato UML entre los distintos tipos de datos reconocidos en el trabajo. La clase ConjuntoTableros mantiene una relación de composición con los tableros según la cual un objeto de tipo ConjuntoTableros está formado por al menos un tablero. La clase Escenario puede estar compuesta por a lo sumo un solo tablero, teniendo en cuenta que en el estado inicial del juego el escenario está vacío (es decir, la cantidad de tableros que lo componen puede reducirse a 0). A su vez, cada tablero está compuesto de un único ConjuntoCasillas, formado por 81 casillas que se clasifican de acuerdo a un valor booleano en pistas y no pistas. Por otro lado, cada Editor posee un único ConjuntoCasillas, que el Editor puede construir a medida que se avanza en el proceso de creación de tableros.

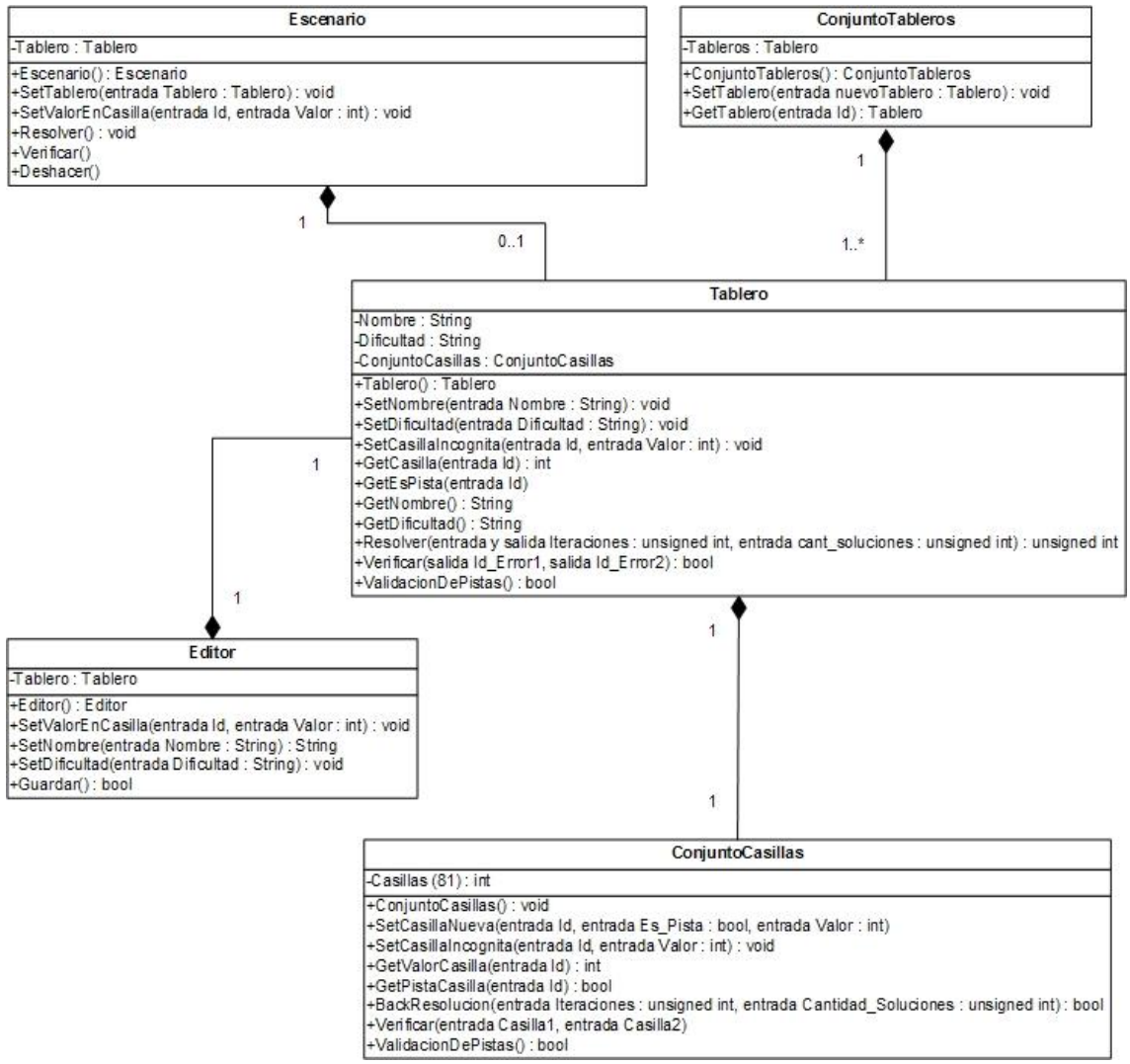

Figura 4: *Diagrama UML de relaciones entre las clases.*

3.2. Carga de tableros

La carga de tableros se realizó a través de una base de datos formada por más de cien sudokus, algunos de ellos previamente investigados y de gran renombre en el campo de la algoritmia. La base de datos no es más que un directorio en la carpeta del programa compuesto por archivos de texto en los que se colocó, utilizando una codificación especial, información que es luego cargada por el programa para rellenar el tablero. El formato de los archivos es tal como el que muestra la *figura 7*. Se utilizó una codificación compleja para evitar que el programa cargue cualquier archivo de texto como si se tratara de un sudoku.

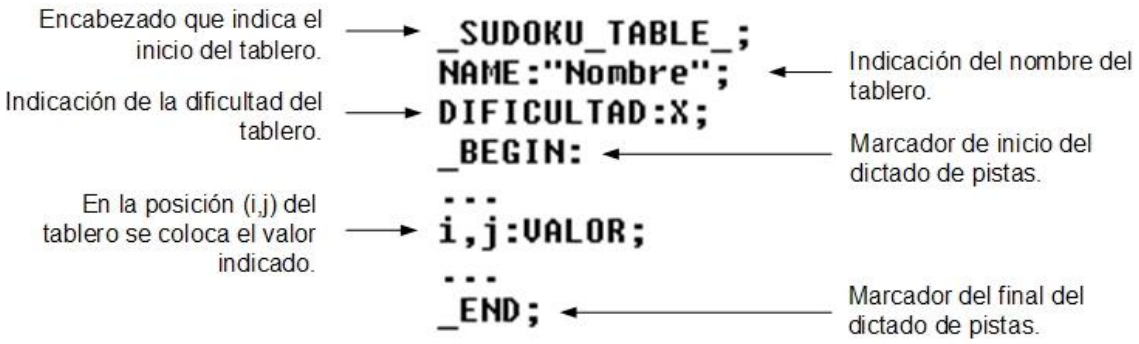

Figura 5: *Sintaxis de los tableros sudokus almacenados en archivos *.txt.*

Ahora bien, un análisis más detallado de la codificación nos permite deducir que ésta puede describirse en términos de un lenguaje regular o de primer orden. Este lenguaje, al que llamaremos L_1, está definido sobre el alfabeto $A = \{0,1,2,3,4,\dots,9,A,B,\dots,Z\}$ más los símbolos ";", "_", ":" y ",". Así, el algoritmo que comprobará la validez de los tableros no es más que la codificación en C++ de la función de transición de estados δ del autómata finito que reconoce el lenguaje. En la *figura 8* se observa dicha función junto a la definición formal del autómata.

Si miramos la función de transición, podemos descomponerla también en términos de sentencias en C++. Esto es:

- Dados dos estados e_i y e_j del autómata unidos por un arco, con $e_i \neq e_j$, el arco puede entenderse como una comprobación acerca de si el valor leído en cinta de entrada (en nuestro caso, el archivo) coincide con el rótulo del arco. En C++ esto puede traducirse como un condicional *If*.
- El arco que une e_i con e_i puede entenderse como un ciclo que dura hasta dar con el valor que rotula el arco que cambia de estado. Esto se entiende fácilmente como un ciclo *While.*
- Llegar a un estado final en el autómata es lo mismo que atravesar exitosamente todo el conjunto de ciclos y de condicionales del código. Al llegar a este estado, el programa está en condiciones de considerar al tablero válido.

- Si el autómata no reconoce una cadena dada (se detiene en un estado no final), significa que algún ciclo se cortó antes de tiempo o que no se cumplió con alguna condición, con lo cual el programa no considerará al tablero como válido.

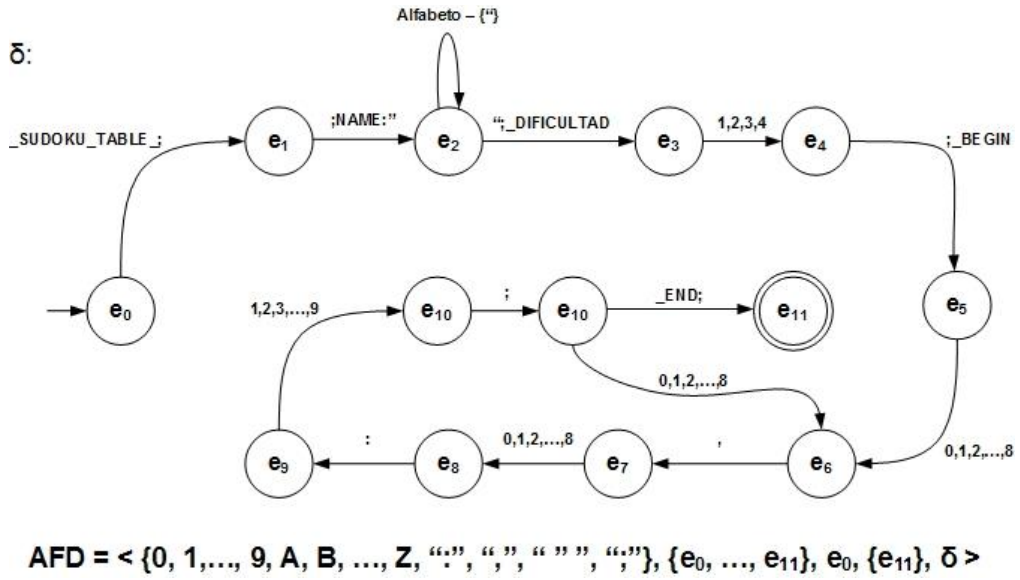

Figura 6: *Autómata finito determinístico que reconoce archivos de texto que contienen sudokus válidos. El algoritmo de reconocimiento no es más que una codificación en C++ de la función de transición de estados.*

Ahora bien, hemos dicho anteriormente que el acceso al disco rígido supone sacrificar un cierto tiempo de ejecución, que disminuye la eficiencia final de la carga. Para evitar eso, se planteó una alternativa que consiste en diseñar una lista de tableros válidos[4], cargados a partir de un directorio raíz al inicio de la aplicación, y seleccionar un elemento de la lista al azar a través de una función aleatoria cada vez que se desee jugar uno nuevo. De esta forma, se cargarán los datos del disco una única vez, y el costo de elegir un elemento de esa lista será de a lo sumo $O(n)$.

En adhesión a esta funcionalidad, se planteó la necesidad de poder abrir tableros que se encontraran almacenados en otros directorios. Para ello, al seleccionar un archivo de texto, se comprobará utilizando el algoritmo de reconocimiento del autómata finito si se trata de un tablero válido, y lo colocará al final de la lista general, de forma tal que en otro instante del juego pueda accederse a él también de manera aleatoria.

3.3. Resolución de tableros

Para resolver los tableros se diseñó un algoritmo de backtracking con podas, lo cual mejora la eficiencia del mismo. Se basa en probar, en orden, los valores del 1 al 9; esto es, coloca el

[4] En este caso se utiliza una lista para representar el TDA ConjuntoTableros, ya que esta clase proporciona los métodos necesarios.

primero de los valores posibles en la primera celda vacía a la que tiene acceso: si la solución parcial es factible (es decir, si ningún valor, incluido el colocado, está repetido en alguna fila, columna o la región de la posición actual), se calcula la siguiente posición modificable y se repiten estos pasos hasta dar con una solución. Sí en algún momento el tablero parcial no es factible, no se coloca ese valor y se intenta colocar el siguiente. Si esto se repite para todos los valores restantes, el backtracking retrocederá hasta el valor inmediato anterior y lo incrementará en uno. El pseudocódigo del algoritmo general del backtracking está disponible en *pseudocódigo 1*, mientras que en *pseudocódigo 2* se encuentran las funciones que éste invoca.

```
void Backtracking(casilla_actual, int solucion_encontrada)
{
    Si (casilla_actual)=?
    {
      valor_actual = 1
      Mientras (solucion_encontrada > 0) && (TengoHijos(valor_actual))
      {
        casilla_actual = valor_actual;
        Si (EsFactible())
        {
            Si (casilla_actual == casilla_final)
               solucion_encontrada--;
            Sino
               Backtracking(siguiente_casilla, solucion_encontrada);
        }
        valor_actual++;
      }
      Si (solucion_encontrada > 0) && (!TengoHijos())
        casilla_actual = ?;
    }
    Sino
    {
      Si (casilla_actual == casilla_final)
        solucion_encontrada--;
      Sino
        Backtracking(siguiente_casilla, solucion_encontrada);
    }
}
```

Pseudocódigo 1: *Pseudocódigo del algoritmo de backtracking que resuelve tableros. La función TengoHijos() se encarga de verificar que el valor_actual sea menor a 10.*

Analicemos el pseudocódigo con mayor detenimiento: el módulo Backtracking recibe al comienzo como parámetros la casilla inicial del tablero (es decir, la $(0,0)$), además de un entero que determina la cantidad de soluciones deseada. Si la casilla actual posee un "?" significa que es modificable, con lo cual puede iniciarse la búsqueda. Mientras la solución no se encuentre y TengoHijos() retorne verdadero (es decir, si los valores a probar oscilan entre 1 y 9), se colocará el valor en la casilla y se verificará si la solución parcial es factible. Si se encuentra un error, el algoritmo probará con el siguiente valor posible; sino, si la casilla actual

es justamente la última casilla, se disminuirá el parámetro solución_encontrada puesto que se ha dado con ella; caso contrario, se continuará ejecutando el backtracking pasándole como parámetro la siguiente posición a ser analizada.

```
bool ErrorDeBloque(int origenX, int destinoX, int origenY, int destinoY)
{
  bool cortar;
  A = {1, 2, 3, 4, 5, 6, 7, 8, 9};
  Para (origenX <= x <= destinoX)
  {
    Para (origenY <= y <= destinoY)
    {
      Si casilla(x,y) Pertenece(A)
        A = A - casilla(x,y);
      Sino
        cortar = true;
    }
  }
  return cortar;
}

bool EsFactible(casilla_actual)
{
  bool factibilidad;
  factibilidad=!(ErrorDeBloque(casilla_actual.fila(),casilla_actual.fila(),0,8));
  Si (factibilidad)
  {
    factibilidad=!(ErrorDeBloque(0,8,casilla_actual.columna(),casilla_actual.columna()));
    Si (factibilidad)
    {
        int i, j;
        i = ComienzoBloque(casilla_actual.fila());
        j = ComienzoBloque(casilla_actual.columna());
        factibilidad = !(ErrorDeBloque(i,i+2,j,j+2));
    }
  }
  return (factibilidad);
}
```

Pseudocódigo 2: *Pseudocódigos de las funciones que invoca el backtracking. La función ErrorDeBloque() devuelve un booleano indicando si existen dos valores repetidos en un bloque dado. Obsérvese que para verificar en filas o en columnas basta con analizar bloques rectangulares de área 1 x 8 u 8 x 1. La función ComienzoBloque() no se escribe dada su proverbial trivialidad. La función EsFactible() devuelve un booleano indicando si no existen valores repetidos en fila, columna o bloque.*

Observemos que la función EsFactible() recibe como parámetro una casilla. A partir de ella, la función verifica que no haya valores repetidos en la fila, la columna o el bloque a los que pertenece la misma, a través del módulo ErrorDeBloque() (entendiendo que la factibilidad se logra sólo si no existen errores en ninguna de las instancias mencionadas). El método ErrorDeBloque() recibe como parámetros los extremos de un rectángulo de análisis (véase *figura 9*); es decir, la función se encargará de analizar que entre los vértices indicados no existan valores repetidos. Sea $A = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ el conjunto de valores que pueden ocupar una de las casillas. El algoritmo quitará del conjunto cada valor a medida que los va leyendo: cuando encuentre que un valor no existe significará que ese valor ya ha sido restado de A (es decir, está repetido). Luego, devolverá verdadero indicando que hubo errores. De lo

contrario, continuará quitando los valores hasta que no existan más posibilidades y se hayan analizado todas las casillas.

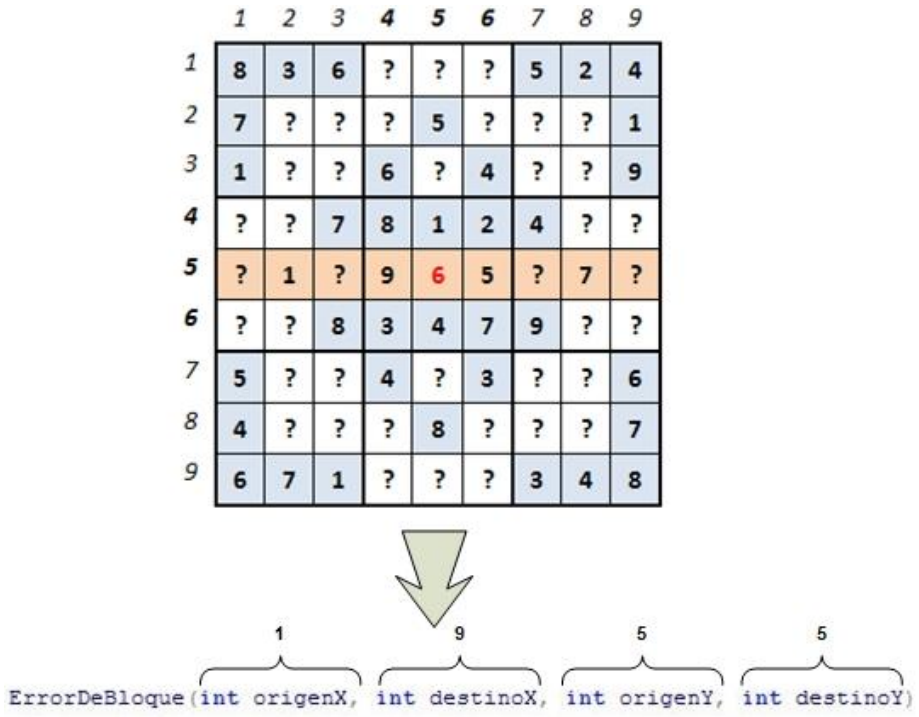

Figura 7: *Un ejemplo de llamado a la función ErrorDeBloque.*

El estudio del costo computacional de la función EsFactible() es fácilmente cuantificable, teniendo en cuenta que está basado en la complejidad temporal de la función ErrorDeBloque(), que es $O(n)$, donde $n = 9$. Así, EsFactible() tiene un costo total de $O(3n) \equiv O(n)$, algo muy aceptable teniendo en cuenta que este algoritmo se utiliza con regularidad en el backtracking.

Una variación de este algoritmo se utiliza a la hora de diseñar tableros con el editor. Como se ha dicho anteriormente, un tablero sudoku es válido sólo si tiene una única solución. Por ello, cuando se intenta guardar un tablero, se ejecuta un backtracking con poda que recorre por completo el espacio de soluciones verificando que no exista más de una sola. El costo computacional promedio de este algoritmo es mayor al del anterior, puesto que no se detiene al hallar la solución sino que continúa la búsqueda hasta que no existan más posibilidades. Así, el costo computacional en el peor de los casos de este algoritmo es $O\left(9^{(9\times9)-17}\right) \approx 1{,}17 \times 10^{61}$.

3.4. Algoritmo de verificación

Otra de las funciones implementadas en el proyecto se encarga de verificar si el tablero activo en pantalla es ya solución o no. Para ello, basta con controlar si en toda fila, en toda columna o

en todo bloque existen valores repetidos, o si no existen celdas marcadas con "?". De verificarse alguna de estas condiciones, el algoritmo devolverá false, indicando que no se ha llegado a la solución. Por el contrario, si ninguna de estas condiciones se cumple, podrá afirmarse que la instancia del juego es una solución.

Así, el algoritmo queda determinado como lo indica el *pseudocódigo 3*.

```
bool Verificar(tablero)
{
    bool cortar = false;
    int i, j;
    i = 0;
    Mientras ((i<9) && (!cortar))
    {
        cortar = ErrorDeBloque(i,i,0,8,false);
        i++;
    }
    i = 0;
    Mientras ((i<9) && (!cortar))
    {
        cortar = ErrorDeBloque(0,8,i,i,false);
        i++;
    }
    i = 0;
    Mientras ((i<3) && (!cortar))
    {
        j = 0;
        Mientras ((j<3) && (!cortar))
        {
            cortar = ErrorDeBloque(i*3, (i*3)+2, j*3, (j*3)+2,false);
            j++;
        }
        i++;
    }
    Si (cortar)
        IndicarError();
    Sino
        IndicarVictoria();
}
```

Pseudocódigo 3: *Algoritmo de verificación, utilizado para determinar si una instancia del juego es solución.*

El costo computacional de este algoritmo depende exclusivamente de la complejidad temporal de las funciones ErrorDeBloque(), que es $O(n)$, donde n es el número de celdas de un bloque (en nuestro caso, 9). De esta forma, claramente el algoritmo de verificación tiene un costo de $O(n \times m)$, con m la cantidad de bloques. En nuestro código, como $n = m = 9$, el costo es $O(n^2)$.

4. Implementación

4.1. Introducción a los tipos de datos implementados

A medida que se avanzó en el análisis del lenguaje utilizado y de las librerías que éste incluye, advertimos que la implementación de algunas clases era trivial e incluso innecesaria. Tal es el caso de, por ejemplo, el tipo de dato Casilla, que ni siquiera se incluyó en la fase de diseño puesto que una casilla es meramente un número entero. Tampoco se implementaron los identificadores de tableros y de casillas (en el primer caso es un número entero, en el segundo un par ordenado) ni el TDA ConjuntoTableros (que es simplemente una lista STL de tableros). Respecto del escenario y el editor, fueron absorbidos totalmente por los formularios[5] del programa, en los cuales se incorporaron funcionalidades extra.

Así, pues, las únicas clases que fueron implementadas son el ConjuntoCasillas y el Tablero, que son la columna vertebral de la aplicación.

A modo de repaso diremos que C++ permite al programador crear un esquema general de los métodos y las estructuras que forman parte de cada clase en archivos .H, y especificar la implementación de cada método en particular a través de los archivos .CPP.

4.2. La clase ConjuntoCasillas

La clase ConjuntoCasillas, como se especificó en la sección 3, está formada básicamente por un conjunto de 81 casillas clasificadas conforme a su comportamiento en pistas y no-pistas o incógnitas. La forma más útil de representar este tipo de dato es mediante matrices: una de enteros, donde se almacenan los valores que cada casilla posee; y otra de booleanos, donde se determina si una celda (i, j) es pista o no. De esa forma, existe una correspondencia biunívoca entre ambas matrices, según la cual cada celda (i, j) de cada matriz hace referencia a la misma celda de la otra matriz (*figura 10*). Los valores que se almacenan en la matriz de enteros oscilan entre 0 y 9: los primeros ocho valores representan los números del conjunto de indistinguibles $\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$; el número restante se identifica con el nombre *Pregunta* y se utiliza como constante en toda la implementación de la clase.

Respecto de los métodos, no existen diferencias sustanciales entre las declaraciones citadas en la sección 4 y las que se implementaron. Analizaremos sin embargo cada uno de ellos en particular, abordando el funcionamiento línea a línea del código fuente final.

[5] Un formulario es una ventana de aplicación. Los formularios son los que permiten que nuestra aplicación tenga una interfaz visual y compatible con Microsoft Windows.

	0	1	2	3	4	5	6	7	8
0	7	2	5	9	9	9	4	1	3
1	6	9	9	9	4	9	9	9	0
2	0	9	9	5	9	3	9	9	8
3	9	9	6	7	0	1	3	9	9
4	9	0	9	8	5	4	9	6	9
5	9	9	7	2	3	6	8	9	9
6	4	9	9	3	9	2	9	9	5
7	3	9	9	9	7	9	9	9	6
8	5	6	0	9	9	9	2	3	7

	0	1	2	3	4	5	6	7	8
0	1	1	1	0	0	0	1	1	1
1	1	0	0	0	1	0	0	0	1
2	1	0	0	1	0	1	0	0	1
3	0	0	1	1	1	1	1	0	0
4	0	1	0	1	1	1	0	1	0
5	0	0	1	1	1	1	1	0	0
6	1	0	0	1	0	1	0	0	1
7	1	0	0	0	1	0	0	0	1
8	1	1	1	0	0	0	1	1	1

Figura 8: *Estados de las matrices Casillas[9][9] y EsPista[9][9] al inicio de un tablero. Obsérvese que las casillas con valores dados están marcadas en TRUE (son pistas) y las que tienen el valor de incógnita 9 no (no son pistas).*

4.2.1. Método constructor

El método constructor de la clase inicializa la matriz Casillas[9][9] con el valor *Pregunta*, y a la matriz EsPista[9][9] en false. De esta forma, una vez construida la clase, se obtienen estructuras que representan claramente un escenario vacío, sin valores.

En *código 1* se aprecia la implementación de este método.

```
ConjuntoCasillas::ConjuntoCasillas()
{
    int i,j;
    for (i=0;i<9;i++)
        for(j=0;j<9;j++)
        {
            Casillas[i][j] = Pregunta;
            EsPista[i][j] = false;
        }
}
```

Código 1: *Implementación del método constructor de la clase ConjuntoCasillas.*

4.2.2. SetCasillaNueva

El método se encarga de colocar un valor dado en una casilla especificada, y determina a su vez si la casilla es o no pista. En la implementación, la función recibe en uno de sus parámetros un par de enteros Id, que representa los índices i y j de la casilla en ambas matrices. Luego, el método asigna en la matriz Casillas el valor entero que recibe como parámetro y en la matriz EsPista el valor booleano, si Valor oscila entre 0 y 8. Si el valor es, por el contrario, 9 (el de la constante *Pregunta*), se marca la casilla como incógnita, sin importar el booleano.

La implementación está expuesta en *código 2*.

```
void ConjuntoCasillas::SetCasillaNueva(pair<int,int> Id, bool Es_Pista, int Valor)
{
    if ((Valor>=0) && (Valor<9))
    {
        Casillas[Id.first][Id.second] = Valor;
        EsPista[Id.first][Id.second] = Es_Pista;
    }
    else if (Valor == 9)
    {
        Casillas[Id.first][Id.second] = Valor;
        EsPista[Id.first][Id.second] = false;
    }
}
```

Código 2: *implementación del método SetCasillaNueva() de la clase ConjuntoCasillas.*

4.2.3. SetCasillaIncognita

A través de este método, la clase puede cambiar valores de una casilla incógnita, en cualquier momento del juego. El método recibe un par de enteros Id como parámetro que identifica a la celda que va a modificarse, y otro entero Valor que representa la cifra que se colocará allí. Luego, el método solo debe controlar que la casilla dada no sea pista y que el valor sea válido.

```
void ConjuntoCasillas::SetCasillaIncognita(pair<int,int> Id, int Valor)
{
    if ((!EsPista[Id.first][Id.second]) && (Valor>=0) && (Valor<=9))
        Casillas[Id.first][Id.second] = Valor;
}
```

Código 3: *Implementación del método SetCasillaIncognita() de la clase ConjuntoCasillas.*

4.2.4. GetValorCasilla y GetEsPista

Analizaremos estos métodos en su conjunto dada su trivialidad. GetValorCasilla() retorna el valor de una casilla dada y GetEsPista() devuelve si la casilla dada es o no pista. Así, pues, los métodos reciben un par de números enteros representando los índices (i, j) de la matriz, a los cuales debe tenerse acceso para obtener el valor de retorno.

La implementación de estos métodos se expone en *código 4*.

```
int ConjuntoCasillas::GetValorCasilla(pair<int,int> Id)
{
    return (Casillas[Id.first][Id.second]);
}

bool ConjuntoCasillas::GetPistaCasilla(pair<int,int> Id)
{
    return (EsPista[Id.first][Id.second]);
}
```

Código 4: *Métodos GetValorCasilla() y GetPistaCasilla() de la clase ConjuntoCasillas.*

4.2.5. BackResolucion

El método ejecuta un algoritmo de backtracking que encuentra la solución al tablero. Para ello recibe como parámetros dos enteros sin signo: uno que representa el número de iteraciones que realizó el backtracking (*Iteraciones*, inicialmente un 0) y otro indicando la cantidad de soluciones que se desean obtener (*cant_soluciones*). El algoritmo avanzará en el espacio de soluciones decrementando el parámetro *cant_soluciones* cada vez que se encuentre una, y cortará cuando el árbol haya sido completado o *cant_soluciones* = 0, y retornará el estado en el que quede esta variable. Así, podemos deducir si un tablero dado es válido o no utilizando este método: si pasamos el parámetro *cant_soluciones* = 2 y el algoritmo retorna un 0, significa que tiene más de una solución, con lo cual el tablero no sirve; si *cant_soluciones* = 2, el algoritmo ya recorrió todo el espacio de soluciones y no pudo decrementar nunca el parámetro, con lo cual no tiene soluciones y tampoco es útil; por otro lado, si cant_soluciones = 1, entonces el tablero es válido puesto que tuvo una única solución.

Para implementar este método se utilizaron otras funcionalidades declaradas como privadas, de tal forma que estén encapsuladas dentro de la estructura y no pueda tenerse acceso a ellas desde una entidad superior. En la *figura 11* se observa la dependencia de este método de los privados.

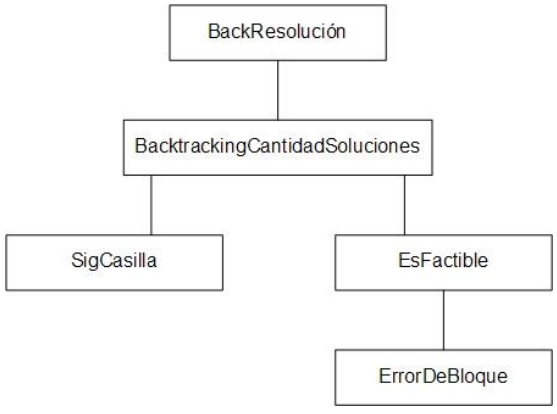

Figura 9: *Diagrama de estructuras*[6] *del método BackResolucion() de la clase ConjuntoCasillas. La función SigCasilla() es la única que no constituye un método privado de la clase, sino un procedimiento auxiliar.*

[6] A ciencia cierta no se trata de un verdadero diagrama de estructuras; el objeto de esta clase de diagramas es simplemente aclarar con mayor énfasis la división procedural que se realizó de los distintos métodos.

El método interno BacktrackingCantidadSoluciones() es el núcleo de la funcionalidad, encargado de ejecutar el backtracking en sí mismo y calcular la cantidad de soluciones obtenidas. Su implementación queda a disposición del lector en *código 5*.

```
void ConjuntoCasillas::BacktrackingCantidadSoluciones(pair<int, int>
CasillaActual, unsigned int & Iteraciones, unsigned int & Cantidad_Soluciones)
{
    if (!EsPista[CasillaActual.first][CasillaActual.second])
    {
        // K es el valor que estamos probando en la matriz
        int k = 0;
        while ((Cantidad_Soluciones>0) && (k<9))
        {
            Iteraciones++;
            Casillas[CasillaActual.first][CasillaActual.second] =  k;
            if (Es_Factible(CasillaActual))
            {
                if ((CasillaActual.second == 8) && (CasillaActual.first == 8))
                    Cantidad_Soluciones--;
                else
                {
                    pair<int,int> nueva_casilla;
                    nueva_casilla = SigCasilla(CasillaActual);
                    BacktrackingCantidadSoluciones(nueva_casilla, Iteraciones,
                    Cantidad_Soluciones);
                }
            }
            k++;
        }
        if ((Cantidad_Soluciones > 0) && (k==9))
            Casillas[CasillaActual.first][CasillaActual.second]=Pregunta;
    }
    else
    {
        if ((CasillaActual.first==8) && (CasillaActual.second==8))
            Cantidad_Soluciones--;
        else
        {
            pair<int,int> nueva_casilla;
            nueva_casilla = SigCasilla(CasillaActual);
            BacktrackingCantidadSoluciones(nueva_casilla, Iteraciones,
            Cantidad_Soluciones);
        }
    }
}
```

Código 5: *Implementación del método privado BacktrackingCantidadSoluciones() de la clase ConjuntoCasillas().*

El algoritmo trabaja de la siguiente manera: dada una casilla (inicialmente, el origen, $(0,0)$), analiza en primera instancia si se trata de una pista; de no ser así, significa que es una casilla incógnita en la que puede probar colocando alguno de los 9 valores del conjunto (para el usuario, $\{1, 2, \dots, 9\}$; para el algoritmo, $\{0, 1, \dots, 8\}$), con lo cual inicializa el contador k en 0 (k es una variable entera que representa el valor probado en la casilla). Mientras la variable *Cantidad_Soluciones* sea mayor que 0 y queden valores posibles para probar en la casilla, el algoritmo verificará si el cambio realizado hace factible a la solución parcial (a través del método privado Es_Factible(), analizado más adelante). Si el cambio se aprueba, se verificará si la casilla es la $(8,8)$ (es decir, la última), pues en tal caso se ha dado con una solución y debe

decrementarse el parámetro *Cantidad_Soluciones*. De no ser así, el algoritmo calcula la siguiente casilla de análisis utilizando la función SigCasilla() e invoca recursivamente al backtracking. Cuando retorne del llamado recursivo, incrementará la variable k para probar el siguiente valor y repetirá todo esto hasta que k no sea un valor válido de casilla. Cuando esto ocurra significa que no existe forma alguna de obtener la solución, con lo cual colocará el valor de incógnita en la casilla nuevamente.

Si la casilla actual fuese una casilla pista, el algoritmo verificará que no se trate de la última casilla, pues en tal caso se ha dado con una solución. De otra forma, se calculará la próxima casilla y se llamará recursivamente al backtracking.

La función interna SigCasilla() (*código 6*) retorna un par de enteros representando la nueva posición de la casilla válida. Para ello, controla si la segunda coordenada es inferior a 8 (esto es, si la casilla sigue dentro de la fila): en tal caso, puede incrementar la coordenada sin ningún problema; si, por el contrario, la segunda coordenada es 8, entonces debemos avanzar hacia la siguiente casilla de la próxima fila, y volver a 0 la segunda coordenada.

```
pair<int, int> SigCasilla(pair<int, int> Actual)
{
    pair<int, int> nueva;
    if (Actual.second<8)
    {
        nueva.first = Actual.first;
        nueva.second = Actual.second + 1;
    }
    else
    {
        nueva.first = Actual.first + 1;
        nueva.second = 0;
    }
    return nueva;
}
```

Código 6: *Implementación de la función interna SigCasilla().*

Respecto del método privado Es_Factible() (*código 7*), se asemeja en gran medida a la funcionalidad explicada en la sección 3.3. Dado un par de enteros que representa la casilla que debe analizarse, el algoritmo controla que no existan repeticiones de valores en la fila, en la columna y el bloque que ocupa. Si encuentra un error, retornará false (la solución parcial no es factible); caso contrario, retornará true.

Obsérvese que utiliza de manera interna el método privado ErrorDeBloque() (*código 8*). Esta funcionalidad recibe como parámetros cuatro números enteros: origenX, origenY, destinoX y destinoY. Estos valores se comportan como los vértices de un rectángulo demarcado en el interior de la matriz, dentro del cual se realiza el análisis para saber si tiene valores repetidos o no.

```
bool ConjuntoCasillas::Es_Factible(pair<int, int> Actual)
{
    bool factibilidad;
    factibilidad = !(ErrorDeBloque(Actual.first,Actual.first,0,8));
    if (factibilidad)
    {
        factibilidad = !(ErrorDeBloque(0,8,Actual.second,Actual.second));
        if (factibilidad)
        {
            int i, j;
            if (Actual.first<3)
                i = 0;
            else if (Actual.first<6)
                i = 3;
            else
                i = 6;
            if (Actual.second<3)
                j = 0;
            else if (Actual.second<6)
                j = 3;
            else
                j = 6;
            factibilidad = !(ErrorDeBloque(i,i+2,j,j+2));
        }
    }
    return (factibilidad);
}
```

Código 7: *Implementación del método privado Es_Factible() de la clase ConjuntoCasillas()*

```
bool ConjuntoCasillas::ErrorDeBloque(int origenX, int destinoX, int origenY, int destinoY)
{
    bool arr[9];
    int i,aux;
    aux = 0;
    for (i=0;i<9;i++)
        arr[i] = false;
    bool cortar = false;
    int k,l;
    k = origenX;
    while ((k<=destinoX) && (!cortar))
    {
        l = origenY;
        while ((l<=destinoY) && (!cortar))
        {
            aux = Casillas[k][l];
            if (aux != Pregunta)
            {
                if (arr[aux]==false)
                    arr[aux]=true;
                else
                    cortar = true;
            }
            l++;
        }
        k++;
    }
    return cortar;
}
```

Código 8: *Implementación del método privado ErrorDeBloque() de la clase ConjuntoCasillas.*

4.2.6. Verificar

El método Verificar() se encarga de controlar si el tablero activo ya es una solución. Si tenemos en cuenta que las casillas se representan utilizando una matriz de enteros, el algoritmo se basa en verificar que en las filas y las columnas de la matriz no existan valores repetidos ni incógnitas, así como también en los 9 bloques que la componen. Utiliza un método privado muy similar al ErrorDeBloque() que se emplea para resolver los tableros, con una pequeña variación para obtener las posiciones de las casillas erróneas y marcarlas en pantalla.

Ahora bien, la función ErrorDeBloqueVerificar() (*código 10*), como se dijo anteriormente, introduce algunas modificaciones respecto del núcleo central, analizado con detenimiento en la sección anterior. Esto es, si *cortar* = true (cuando se ha encontrado algún error), el algoritmo busca las casillas erróneas en el tablero y retorna sus posiciones en los pares Casilla1 y Casilla2 que recibe como parámetros.

```
bool ConjuntoCasillas::Verificar(pair<int, int> & Casilla1, pair<int, int> & Casilla2)
{
    bool cortar = false;
    int i, j;
    i = j = 0;
    // Analizamos las columnas
    while ((i<9) && (!cortar))
    {
        cortar = ErrorDeBloqueVerificar(i,i,0,8, Casilla1, Casilla2);
        i++;
    }
    // Analizamos las filas
    while ((j<9) && (!cortar))
    {
        cortar = ErrorDeBloqueVerificar(0,8,j,j, Casilla1, Casilla2);
        j++;
    }
    // Analizamos los bloques
    i = 0;
    while ((i<3) && (!cortar))
    {
        j = 0;
        while ((j<3) && (!cortar))
        {
            cortar = ErrorDeBloqueVerificar(i*3, (i*3)+2, j*3, (j*3)+2, Casilla1, Casilla2);
            j++;
        }
        i++;
    }

    return (!cortar);
}
```

Código 9: *Implementación del método Verificar() de la clase ConjuntoCasillas.*

```
bool ConjuntoCasillas::ErrorDeBloqueVerificar(int origenX, int destinoX, int
origenY, int destinoY, pair<int, int> & Casilla1, pair<int, int> & Casilla2)
{
    bool arr[9];
    int i;
    for (i=0;i<9;i++)
        arr[i] = false;
    bool cortar = false;
    int k,l;
    k = origenX;
```

```
    while ((k<=destinoX) && (!cortar))
    {
        l = origenY;
        while ((l<=destinoY) && (!cortar))
        {
            if (Casillas[k][l]!=Pregunta)
            {
                if (arr[Casillas[k][l]]==false)
                    arr[Casillas[k][l]]=true;
                else
                    cortar = true;
            }
            else
                cortar = true;
            l++;
        }
        k++;
    }
    if (cortar)
    {
        Casilla1.first = k-1;
        Casilla1.second = l-1;
        if (Casillas[Casilla1.first][Casilla1.second] != Pregunta)
        {
            k = origenX;
            l = origenY;
            bool encontrado = false;
            while ((k<=destinoX) && (!encontrado))
            {
                while ((l<=destinoY) && (!encontrado))
                {
                    if (Casillas[Casilla1.first][Casilla1.second] ==
                    Casillas[k][l])
                    {
                        encontrado = true;
                        Casilla2.first = k;
                        Casilla2.second = l;
                    }
                    l++;
                }
                k++;
            }
        }
        else
            Casilla2 = Casilla1;
    }
    return cortar;
}
```

Código 10: *Implementación del método privado ErrorDeBloqueVerificar() de la clase ConjuntoCasillas.*

4.2.7. ValidacionDePistas

El método ValidacionDePistas() se utiliza en la clase ConjuntoCasillas para verificar que las pistas dadas de antemano no estén repetidas en fila, columna o bloque. De esta forma, si los tableros son modificados por error directamente sobre el archivo .TXT, se ahorrará el recorrido innecesario del backtracking y se colocará un mensaje de error en pantalla. Por otra parte, durante la edición de tableros, el método ahorrará al algoritmo de backtracking todos sus recorridos en caso de que el usuario pretenda guardar un tablero con dos o más casillas repetidas. Obsérvese en *código 11* que no presenta diferencias sustanciales respecto del

método Verificar(): simplemente llama a ErrorDeBloque() en lugar de a ErrorDeBloque Verificar() para encontrar las coincidencias erróneas.

```
bool ConjuntoCasillas::ValidacionDePistas()
{
    int i,j;
    for (i=0;i<9;i++)
        for (j=0;j<9;j++)
            if (!EsPista[i][j])
                Casillas[i][j]=Pregunta;
    bool cortar = false;
    i = j = 0;
    // Analizamos las columnas
    while ((i<9) && (!cortar))
    {
        cortar = ErrorDeBloque(i,i,0,8);
        i++;
    }
    // Analizamos las filas
    while ((j<9) && (!cortar))
    {
        cortar = ErrorDeBloque(0,8,j,j);
        j++;
    }
    // Analizamos los bloques
    i = 0;
    while ((i<3) && (!cortar))
    {
        j = 0;
        while ((j<3) && (!cortar))
        {
            cortar = ErrorDeBloque(i*3, (i*3)+2, j*3, (j*3)+2);
            j++;
        }
        i++;
    }
    return (!cortar);
}
```

Código 11: *Implementación del método ValidacionDePistas() de la clase ConjuntoCasillas.*

4.3. La clase Tablero

La clase Tablero encapsula en cierta forma a la clase ConjuntoCasillas, y le incorpora algunas funcionalidades extra para colocarle el nombre y el nivel de dificultad, e incluso para poder recuperarlos. Analizaremos en detalle sólo aquellos métodos cuya implementación requiere de una explicación compleja, teniendo en cuenta que muchas de las funcionalidades de la clase Tablero solo invocan a los métodos de ConjuntoCasillas.

4.3.1. Método constructor

El método constructor debe crear la interfaz para que pueda comenzar a llenarse la clase con la información pertinente (nombre, dificultad y ConjuntoCasillas), con lo cual genera un puntero a la estructura ConjuntoCasillas, de forma tal que esté listo para poder modificarse. La implementación de esta funcionalidad está dada en *código 12*.

```
Tablero::Tablero()
{
    Casillas = new ConjuntoCasillas();
}
```

Código 12: *Implementación del método constructor de la clase Tablero.*

4.3.2. Método destructor

Así como el método constructor crea el espacio para empezar a llenar con casillas, el método destructor se encarga de liberar la memoria que utiliza el puntero y desreferenciarlo a NULL. La implementación puede verse en detalle en *código 13*.

```
Tablero::~Tablero()
{
    delete Casillas;
    Casillas = NULL;
}
```

Código 13: *Implementación del método destructor de la clase Tablero.*

4.3.3. SetNombre y SetDificultad

Los métodos SetNombre() y SetDificultad() se encargan de asignarle al tablero un nombre y un nivel de dificultad, que reciben como parámetros de tipo string. Los códigos fuente de ambos métodos están disponibles en *código 14*.

```
void Tablero::SetNombre(string nombre)
{
    Nombre = nombre;
}

void Tablero::SetDificultad(string difcultad)
{
    Dificultad = difcultad;
}
```

Código 14: *Implementación de los métodos SetNombre() y SetDificultad() de la clase Tablero.*

4.3.4. GetNombre y GetDificultad

GetNombre() y GetDificultad() (*código 15*) son métodos de la clase que retornan los contenidos de las variables Nombre y Dificultad. Su implementación es trivial: simplemente devuelve los valores almacenados.

```
string Tablero::GetNombre()
{
    return (Nombre);
}

string Tablero::GetDificultad()
{
    return (Dificultad);
}
```

Código 15: *Implementación de los métodos GetNombre() y GetDificultad() de la clase Tablero.*

4.4. Otros algoritmos implementados

Además de haberse implementado los métodos internos de las clases, se implementó un algoritmo de backtracking extra que trabaja de manera similar al método Backtracking CantidadSoluciones(), sólo que muestra por pantalla la exploración del árbol durante la búsqueda. Es decir, el usuario podrá ver en las celdas del tablero cómo la computadora procesa el análisis. Este algoritmo se analizará en detalle en la sección 4.6.

4.5. Implementación de los formularios

La aplicación final trabaja en entorno gráfico, con lo cual la principal herramienta con la que contamos para hacerla atractiva y adecuarla al aspecto de otros programas Windows es el formulario. Un formulario es una ventana clásica, formada por una barra de título y botones de minimización, maximización/restauración y cierre. Para nuestra aplicación, hemos incorporado al formulario inicial de resolución de tableros botones de comando que representan cada casilla del tablero y para accionar las diferentes operaciones, además de una barra de menús como otra alternativa para la ejecución de acciones. De esta forma, la clase Escenario está encapsulada dentro del formulario inicial de juego (de ahí que no ha sido implementada por separado), mientras que la clase Editor permanece contenida en el formulario de edición de tableros. Se incorporan, además, cuadros de diálogo (formularios con funcionalidades disminuidas) para la configuración de los algoritmos de resolución, la introducción de valores en las casillas del tablero, el clásico “Acerca de… ” y los mensajes de error.

El aspecto del formulario utilizado para resolver tableros está expuesto en la *figura 12*. Para una mayor especificación respecto del funcionamiento de cada control[7], consúltese el manual de usuario que se adjunta al programa.

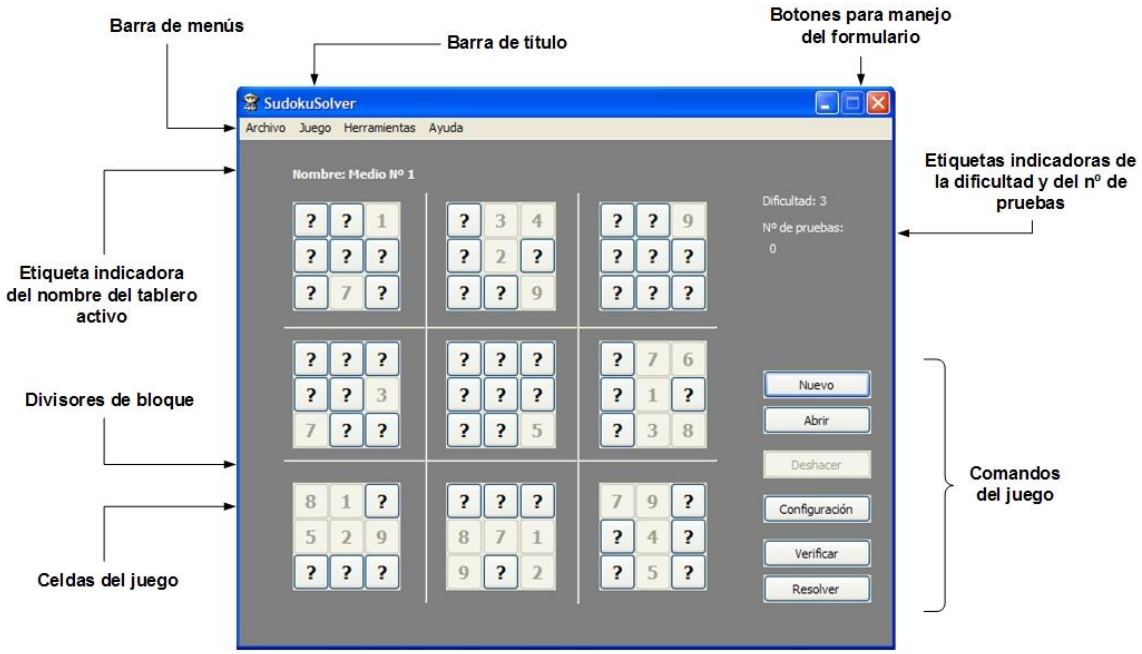

Figura 10: *Formulario inicial del juego, desde donde se resuelven los tableros.*

A este formulario inicial se le incorporaron, además de los controles, algunas propiedades que revisten una cierta importancia, puesto que contienen aquellos atributos de la clase Escenario que, como es obvio, no vienen incorporadas por defecto en él. Las estructuras que se agregaron son:

- Una matriz de botones de comandos (wxTablero) de dimensión 9×9 que representa cada celda del tablero. Cabe considerar que estos botones se crean en tiempo de ejecución, puesto que el IDE utilizado para programar la aplicación no permite la creación de matrices o arreglos de controles en tiempo de diseño.
- Una matriz de enteros (matriz_coordenadas) de dimensión 9×9 en la cual se colocan las coordenadas necesarias para, en tiempo de ejecución, poder situar en pantalla los botones de comando de la matriz wxTablero.
- Un string (Ruta) que representa la ruta por defecto desde la cual el programa carga los tableros (es decir, "\Tableros", perteneciente al directorio de la aplicación).
- Una lista de punteros a tableros (listaTableros), que encapsula en sí misma las funcionalidades de la clase ConjuntoTableros (razón por la cual no ha sido implementada). En ella se colocan todos los tableros del directorio señalado por Ruta, así como también aquellos que el usuario abra durante el transcurso del juego.

[7] Se llama control a cada componente gráfica del formulario: botones de comando, barra de menús, etc.

- Un par formado por un puntero a un botón de comando y un string (cambio_anterior), donde se almacenarán el puntero al botón (celda) donde se realizó el último cambio y el valor que dicho botón tenía antes de hecha la modificación. De esta forma, se podrá deshacer el último cambio hecho con facilidad.
- Dos strings (error1 y error2) donde el algoritmo de verificación guardará temporalmente el contenido de las posiciones erróneas, de forma tal que, pasados dos segundos luego de señalado el error, puedan restaurarse.[8]
- Dos pares de enteros (pos_erronea1 y pos_erronea2) donde el algoritmo de verificación guardará las posiciones de las casillas donde se encontraron errores.
- Un string (RutaRecursos) donde permanece almacenada la ruta a partir de la cual el programa tiene acceso a los distintos sonidos que se ejecutan en el juego.
- Un boolean (MostrarProgreso) que indica si el usuario desea que la computadora resuelva su tablero utilizando el algoritmo de resolución con muestra del progreso o sin ella.
- Un entero sin signo (Iteraciones), donde permanece almacenada la cantidad de iteraciones que necesitó el algoritmo para encontrar la solución o bien la cantidad de pasos que va realizando el usuario.
- Un puntero a un objeto de la clase Tablero (tablero_actual), donde permanece almacenado el tablero que está activo en pantalla.

Respecto del formulario para la edición de tableros (*figura 13*), también se han incorporado atributos para adaptarlo al funcionamiento de la clase Editor, que no ha sido implementada. Estos son:

- Una matriz de enteros (matriz_coordenadas) de dimensión 9×9, idéntica a la del formulario del juego.
- Una matriz de listas desplegables o *combo boxes* (wxEditores) de dimensión 9×9, donde el usuario colocará las pistas que desea que tenga su tablero. Esta matriz se crea y ubica en pantalla en tiempo de ejecución.
- Un string (RutaTableros) donde el cuadro de diálogo para guardar los tableros se ubicará inicialmente.
- Un puntero a un objeto de la clase Tablero (tablero_actual) donde se creará el tablero al guardar y se cotejará la validez del mismo, antes de crear un archivo.

[8] Se implementan de manera global puesto que el evento Time del control Timer (encargado de la restauración de estos valores) no podría tener acceso a ellos de otra forma.

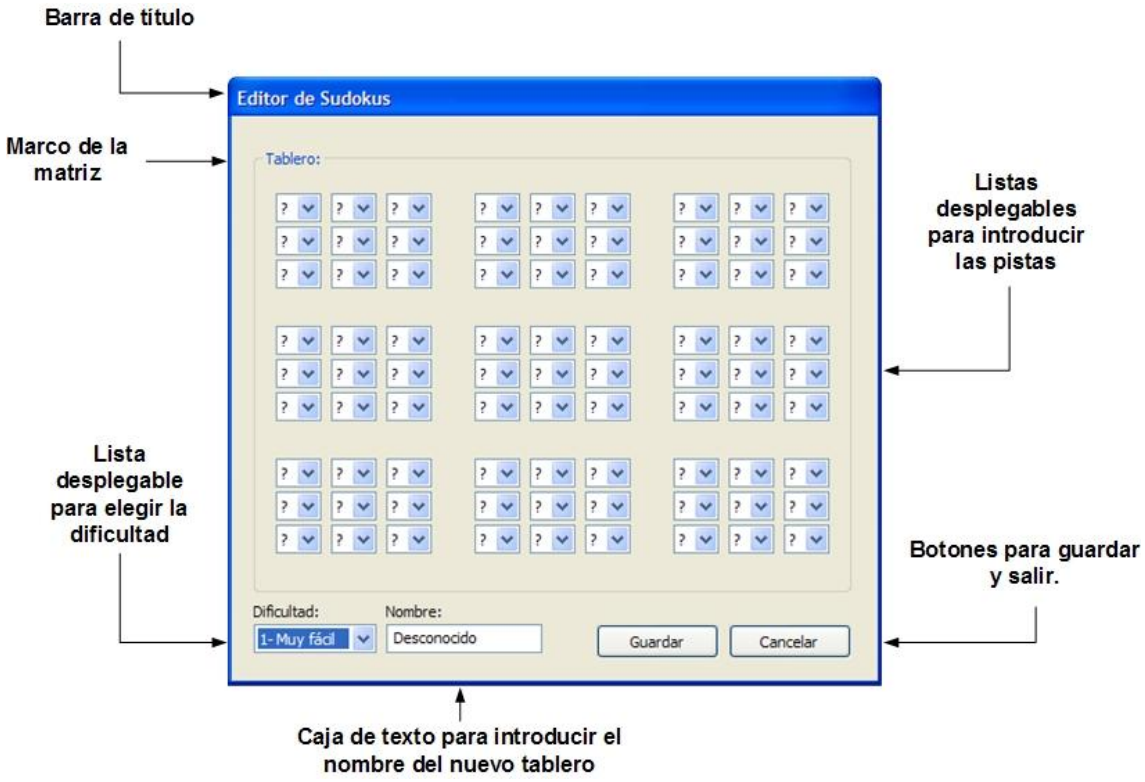

Figura 11: *Formulario de edición de tableros.*

4.6. Implementación de las funcionalidades de SudokuFrm

El formulario inicial que en el código se identifica como SudokuFrm posee muchas funcionalidades propias de la clase Escenario, así como también otras extras que fueron necesarias para la finalización de la aplicación. Las analizaremos en detalle a continuación:

4.6.1. Elección al azar de tableros

La elección al azar de los tableros se realiza cada vez que el usuario presiona en el botón cmdNuevo. La funcionalidad utiliza algunos procedimientos internos cuyo análisis se realizará en casos de extrema necesidad, teniendo en cuenta la gran cantidad de métodos auxiliares que se han implementado. En la *figura 14* se observa un diagrama de estructuras de la funcionalidad en análisis.

Dada la lista de tableros, el algoritmo que se ejecuta al cliquear en el botón cmdNuevo habilitará inicialmente los botones necesarios para poder comenzar a resolver el juego (utilizando AccionarControles, un método privado del formulario SudokuFrm). *A posteriori*, se generará aleatoriamente un número entero entre 1 y la longitud de la lista, que servirá como identificador del tablero cuya resolución se planea iniciar. Para tener acceso a él, se iterará sobre la lista tantas veces como el número indique, y se llamará al método privado GenerarEspacioJuego() pasando como parámetro el iterador de la estructura. Esta función se encarga de asignar al puntero tablero_actual el nuevo tablero, al cual debe asignársele el

conjunto de casillas (que es a su vez expuesto en los botones de comando que representan las celdas), el nombre y el nivel de dificultad. Por último, se crea un objeto de la clase wxSound asociado al sonido de “gong”, y se reproduce utilizando la funcionalidad ReproducirSonido.

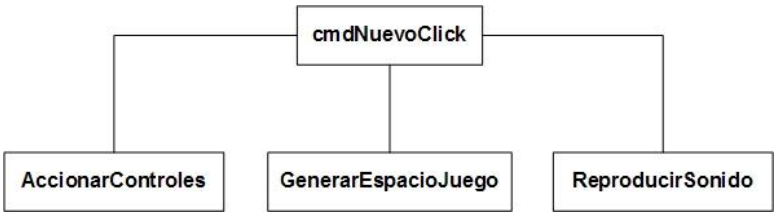

Figura 12: *Diagrama de estructuras del evento cmdNuevoClick() del formulario SudokuFrm.*

4.6.2. Apertura de tableros a elección

Esta funcionalidad se encuentra implementada en el evento cmdAbrirClick(), que se ejecuta al hacer clic en el botón cmdAbrir. Un pequeño diagrama de estructuras puede verse en la *figura 15*.

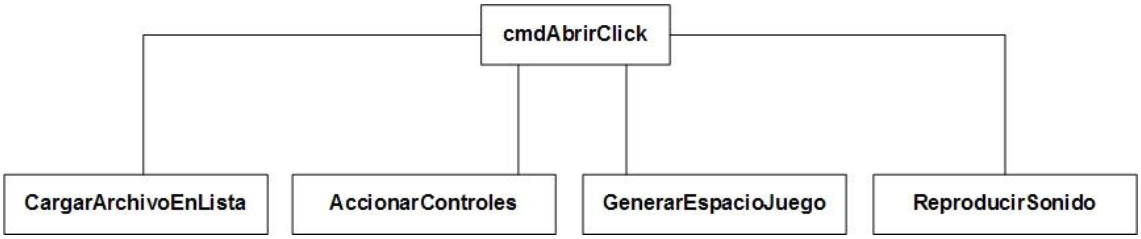

Figura 13: *Diagrama de estructuras del evento cmdAbrirClick() del formulario SudokuFrm.*

Inicialmente, el algoritmo muestra por pantalla un cuadro de diálogo para elegir un archivo .TXT de cualquier directorio del disco (idéntico al de la mayoría de las aplicaciones Windows conocidas). Si se pulsa “Abrir”, se recuperan la ruta donde está el tablero y la longitud actual de la lista de tableros, para luego invocar al método CargarArchivoEnLista() pasándole el path mencionado anteriormente. Este método verificará que la codificación del archivo se corresponda con la de los tableros y, de ser así, lo encolará al final de la lista. Luego, se comparará si la longitud de lista recuperada con anterioridad es distinta a la de la lista nueva: en tal caso, el tablero ha sido cargado con éxito y la aplicación ejecuta las mismas acciones que lleva a cabo el evento cmdNuevoClick() luego de elegido el tablero al azar; caso contrario, significa que el archivo no contenía un tablero válido, con lo cual se muestra un mensaje por pantalla.

4.6.3. Resolución del tablero activo en pantalla

Se encuentra encapsulado en el evento cmdResolverClick(), que tiene lugar cuando se hace clic en el botón cmdResolver(). Se omite el diagrama de estructuras en este caso por cuestiones de espacio.

Al comienzo del algoritmo, se desactivan todos los controles (incluso las celdas del tablero) de forma tal que el usuario no pueda interferir en la libre ejecución de la búsqueda. Teniendo en cuenta que se expone por pantalla el número de iteraciones que demoró al backtracking en encontrar la solución, la etiqueta donde figura la cantidad de pruebas que realizó el usuario cambiará de modo tal que se muestre sólo el número de iteraciones. Ahora bien, sabemos que los tableros que ejecutamos en el programa han sido creados mediante el editor de tableros incorporado, con lo cual tienen siempre una y sólo una solución; sin embargo, se implementó una variación en el algoritmo de forma tal que se controle si el tablero tiene o no solución. Inicialmente se verifica que las pistas del tablero activo no generan contradicciones (es decir, repeticiones de valores en fila, columna o bloque) a través del método PistasValidas() de la clase Tablero. Si se eligió el algoritmo que muestra el progreso de la búsqueda, se restaura en pantalla el tablero a su forma original (es decir, se colocan signos "?" en las casillas que se sabe que no son pista) y se llama al método interno del SudokuFrm BacktrackingSimple(), que busca la solución utilizando los propios botones del formulario. Así, el usuario puede ver los cambios que el algoritmo realiza en las celdas durante el proceso de búsqueda. Además, el algoritmo devuelve un booleano (aux) indicando si se dio con la solución o no, para mostrar (en caso negativo) un mensaje de error por pantalla.

Si se decidió no mostrar la búsqueda, simplemente basta con llamar al método Resolver() de tablero_actual y actualizar los valores de los botones obteniéndolos a partir del tablero resuelto. *A posteriori* se restauran los valores en tablero_actual de forma tal que su conjunto de casillas vuelva al estado inicial en el que se encontraba.

Si se encontró la solución, se reproduce el sonido "tada" y se muestra por pantalla el número de iteraciones realizadas.

4.6.4. Abrir el cuadro de diálogo de configuración

Esta funcionalidad se activa al ejecutar el evento cmdConfiguracionClick() del botón cmdConfiguración. El modo de trabajo es muy sencillo: inicialmente activa los botones de opción de acuerdo al estado de la variable MostrarProgreso (es decir, si MostrarProgreso = false se setea la primer opción; sino, la segunda), se crea un objeto de tipo CuadroDialogoConfiguracion (que es el nombre con el cuál se identifica este cuadro de diálogo) y ejecuta el método ShowModal() (heredado de la clase wxDialog), que permite mostrarlo por pantalla y esperar un valor de retorno. Dependiendo de la opción elegida, el valor de retorno puede ser un 6 (se eligió cancelar), un 1 (se eligió no mostrar el progreso) o un 2 (se eligió mostrarlo). A partir de ellos, el algoritmo cambia el estado de MostrarProgreso según corresponda.

4.6.5. Apertura del editor de tableros

Se ejecuta al seleccionar la opción del menú "Editor de tableros…" (método Mnueditordetableros1036Click()), y su implementación es trivial: se crea un objeto de tipo Editor, se setea el atributo RutaTableros, a través del método público Editor:: SetRutaTablerosEditor(), en Ruta; y se desreferencia el puntero al objeto tablero_actual en edición del Editor, a través del método Editor::AnularTablero().

Un diagrama de estructuras de esta funcionalidad puede verse en la *figura 16*.

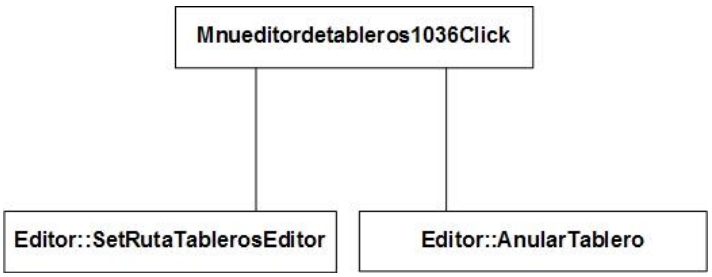

Figura 14: *Diagrama de estructuras del evento Mnueditordetableros1036Click() del formulario SudokuFrm.*

4.6.6. Apertura del cuadro de diálogo "Acerca de"

Al seleccionar la opción del menú "Acerca de", se abre un cuadro de diálogo con información sobre la aplicación. El evento está contenido en el módulo Mnuacercadesudokusolver 1034Click(), y consiste simplemente en reproducir el sonido "curve", crear un objeto del tipo "AcercaDe" e invocar al método ShowModal().

4.6.7. Cierre de la aplicación

Cuando se cliquea en la opción del menú "Salir" se ejecuta el evento Mnusalir1039Click(), que muestra un diálogo por pantalla pidiendo la confirmación de la salida y, de ser aceptado éste, destruye el objeto, cerrándose por consiguiente la aplicación.

4.6.8. Deshacer el último cambio realizado

Teniendo en cuenta que sólo se permite deshacer un único cambio, el algoritmo bloquea el botón cmdDeshacer luego de hacerle clic. Posteriormente, toma el botón contenido en la primera coordenada del par cambio_anterior y le asigna como etiqueta la segunda coordenada, que contiene el string con el último cambio realizado.

Esta funcionalidad está contenida en los eventos cmdDeshacerClick() y MnuDeshacerClick().

4.6.9. Cambio de los valores de las celdas del tablero

Permite al usuario introducir los valores en el tablero (tanto el referenciado por el puntero tablero_actual como el que permanece implícito en pantalla a través de los botones). Se ejecuta al hacer clic en cualquiera de los botones del tablero, lo cual supone la ejecución del evento wxTableroClick(). El algoritmo pedirá por pantalla el valor mientras el usuario ingrese valores erróneos y haga clic en "Aceptar" en el mensaje de alerta que aparece en tal caso (si esto sucede, la variable dato_valido = false), a través del cuadro de ingreso de texto wxIngresarNumero. Los valores permitidos por éste son los números del 1 al 9, además del signo "?": cuando sean ingresados, se incrementará el contador de iteraciones, se actualizará el valor de Iteraciones en pantalla, se conservará el estado anterior del botón pulsado y se lo actualizará con el valor elegido.

4.6.10. Verificación de la validez como solución del tablero en pantalla

Para verificar la validez de los tableros, inicialmente se actualiza tablero_actual con los valores colocados en la matriz wxTablero. Luego, invoca al método Tablero::Verificar(), pasándole como parámetros dos pares de enteros en los que, de existir un error, se guardarán las coordenadas de las casillas que lo cometen. En caso de que Verificar() devuelva un booleano en false, significa que existen errores, con lo cual las celdas erróneas se marcan con una "X", se escucha un sonido y se activa un Timer para recuperar los valores después de 1,5 segundos. Por otro lado, si el booleano es true, el tablero en pantalla es la solución, con lo cual se muestra un mensaje felicitando al jugador y se reproduce el sonido de victoria.

En la *figura 17* puede observarse un pequeño diagrama de estructuras de la funcionalidad.

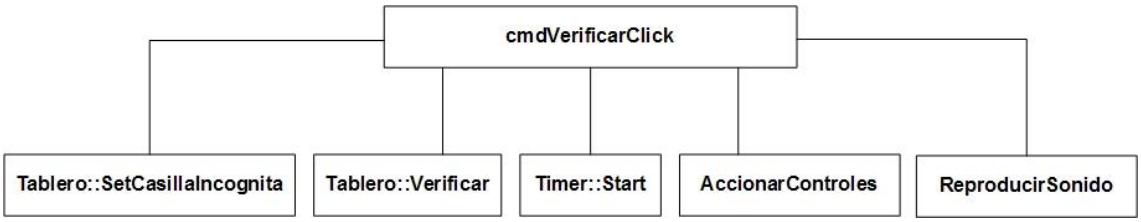

Figura 15: *Diagrama de estructuras del evento cmdVerificarClick() del formulario SudokuFrm.*

4.6.11. Setear la ruta de recursos y de los tableros

Se incorporaron dos funcionalidades para setear rutas: SetRutaRecursos() se encarga de seleccionar la carpeta donde se encuentran los archivos de sonido, y SetRutaTableros() elige el directorio raíz donde se encuentran almacenados los tableros que vienen dados con la aplicación. Según nuestro diseño, los recursos están en la carpeta "\Recursos" y los tableros en "\Tableros", ambos contenidos en el directorio del ejecutable.

Se utiliza un tipo de dato provisto por el IDE, el wxStandardPaths, que encapsula los directorios más importantes del disco. En este caso, basta con llamar al método wxStandardPaths:: GetExecutablePath(), que devuelve la dirección donde está almacenado el archivo ejecutable, y concatenarle con el operador "+" para cadenas de caracteres el nombre de la carpeta que corresponda (es decir, "\Recursos" o "\Tableros").

4.6.12. Determinación del tipo de algoritmo de búsqueda

La implementación del método SetProgreso(), encargado de realizar esta funcionalidad, es trivial: simplemente asigna un booleano que recibe como parámetro al atributo del formulario "MostrarProgreso".

4.6.13. Construcción de la lista de tableros

Al iniciar la aplicación, se ejecuta el método ConstruirListaTableros(), que se encarga de crear lo que en la fase de diseño se entendía como el TDA ConjuntoTableros. Inicialmente, verifica si el directorio de tableros existe (si no existiera, aparecería un error fatal por pantalla y se pediría la reinstalación del juego o la restauración de la carpeta "\Tableros"), para luego crear un iterador que recorra los archivos allí guardados y los cargue en lista usando el método privado CargarEnLista(). Esta funcionalidad permite analizar, en primera instancia, si la extensión del archivo es la correcta (.TXT), y luego invoca a la implementación del autómata finito determinístico reconocedor de tableros, codificada en el procedimiento CargarArchivoEnLista(). Si se llega al estado final, el cual se alcanza si el archivo respeta la codificación estandarizada, se crea el objeto de tipo Tablero y se lo encola en la lista. En caso de que la lista quede vacía, significa que ningún archivo fue reconocido por el autómata, con lo cual se mostrará un error fatal por pantalla, puesto que en el directorio de tableros no está ninguno de los provistos por el programa.

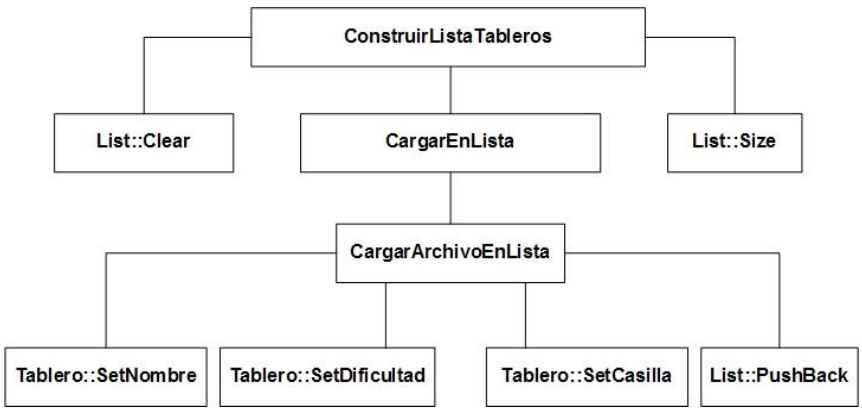

Figura 16: *Diagrama de estructuras del método ConstruirListaTableros() del formulario SudokuFrm.*

4.7. Implementación de las funcionalidades de Editor

El único método del formulario Editor cuya implementación merece un estudio en profundidad es wxGuardarClick(), que se ejecuta al hacer clic en el botón wxGuardar.

El algoritmo inicialmente actualiza tablero_actual con las pistas dadas por el usuario en la matriz wxEditores, formada por listas desplegables, y las cuenta para luego verificar que sean más de 16. *A posteriori*, se utiliza el método Tablero::PistasValidas(), que retorna un booleano indicando que no haya repeticiones de valores en filas, columnas o bloques. Si devuelve true, se llama al método Tablero::Resolver() pasándole como parámetro cant_soluciones = 2. De esta forma, si el tablero es válido, se mostrará un cuadro de diálogo por pantalla para que el usuario seleccione el directorio donde quiere almacenar el archivo y le coloque un nombre, y luego se codificará el tablero para posteriormente guardarlo.

Obsérvese que aquí, a diferencia de en el SudokuFrm, se utilizó otra librería para el manejo de archivos (*fstream* en lugar de *cstdio*). Esto se debe a que la librería *boost* utilizada para el manejo de directorios en el formulario del juego crea inconvenientes con *fstream*.

Un diagrama de estructuras puede analizarse en la *figura 19*.

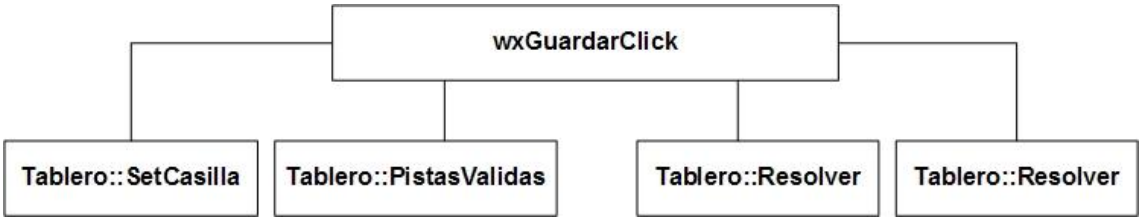

Figura 17: *Diagrama de estructuras para el evento wxGuardarClick() del formulario Editor.*

5. Conclusiones

5.1. Estudio teórico de la eficiencia del algoritmo de resolución

Los algoritmos de backtracking se caracterizan por permitirnos obtener la solución a todo problema de satisfacción de restricciones o de optimización. El costo computacional, sin embargo, está acotado en $O(n^k)$, con n representando la cantidad de hijos de cada nodo y k la cantidad de niveles del espacio de soluciones.

Como se ha dicho antes, nuestros tableros sudokus están formados por 81 casillas, y hasta hoy se han descubierto sudokus que pueden ser resueltos por el ser humano si tienen un mínimo de 17 pistas. Si consideramos que n es la cantidad de casillas modificables de un tablero y que k representa la cantidad de valores posibles que pueden ocupar una celda, el costo final de un algoritmo de backtracking sin poda es $O((9)^{81-17})$.

Ahora bien, las podas implementadas en nuestro algoritmo de backtracking nos permiten reducir la complejidad temporal significativamente, de acuerdo a un criterio que depende de forma exclusiva de las características de los tableros. Esto es, cuanto mayor sea la densidad de pistas en las primeras celdas del sudoku, el número de iteraciones se disminuirá. De la misma forma, si los primeros valores de la solución son altos y no están dados como pistas, el número de iteraciones necesarias se incrementará.

1. Near Worst Case

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

Iteraciones: 622.577.597

2. Con 17 pistas

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

Iteraciones: 239.312.393

Figura 18: *Tableros de ejemplo utilizados para estudiar el número de iteraciones del algoritmo de backtracking.*

Tomemos por ejemplo el tablero conocido como *Near Worst Case* [8] (*figura 20.1*). Su resolución utilizando nuestro algoritmo de backtracking con poda demora un total de 622.577.597 iteraciones debido a la ausencia de pistas en la primera fila y a la densidad de números grandes en ella, teniendo en cuenta que la primera fila es 987654321. En contraposición, el tablero *Con 17 pistas* (*figura 20.2)* demora 239.312.393 iteraciones, un

número considerablemente inferior al del *Near Worst Case*. Su principal diferencia respecto de éste es que tiene una pista en la primera fila, y que la primera fila de la solución es 693784512.

Las podas, por lo tanto, permiten una gran optimización del algoritmo final.

5.2. Estudio empírico de la eficiencia del algoritmo de resolución

A partir de 32 tableros elegidos al azar de la población de tableros propuestos, se realizaron estudios del tiempo de ejecución y del número de iteraciones que demoró el algoritmo de backtracking con poda en dar con los resultados. Estos datos se obtuvieron al correr el programa en una notebook Banghó con un procesador Intel Celeron a 1.99 GHz, con 896 MB de memoria RAM, cronometrados manualmente, y fueron tabulados en las *tablas 1 y 2*, ubicadas al final de la presente sección.

Antes de analizar los tiempos de demora de cada algoritmo, analicemos con detenimiento los resultados de la *Tabla 1*. Si realizamos un muestreo por intervalos de la población, concentrándonos en el número de iteraciones, y dividiéndolos en regiones cada 3.000.000 de iteraciones, podemos ver que en el primer intervalo (desde *Trivial* a *Golden Nugget* inclusive), el 60% de los tableros tiene una densidad baja de números incógnita altos en la primera fila. Esto se debe a que, como se especificó en la sección 5.1, el algoritmo incrementa su eficiencia cuanto menores sean los números que ocupan las primeras filas. Si tomamos todos los intervalos restantes y los agrupamos en uno (desde *Star Burst Leo* hasta *Near Worst Case* inclusive), más del 70% son tableros con una densidad alta de valores incógnita altos en la primera fila.

Por otro lado, hemos dicho que la eficiencia se mejora cuanto mayor sea el número de pistas del tablero. Si los clasificamos de acuerdo al número de pistas, podemos notar que los que tienen la cantidad mínima de 17 (*Near Worst Case* y *Con 17 pistas*) demoran un gran número de iteraciones en ser resueltos, mientras que, conforme aumenta la cantidad de pistas, la población se comporta (en promedio) de manera esperada (es decir, el número de iteraciones disminuye).

Respecto de los datos compilados en *Tabla 2*, puede afirmarse que el algoritmo de backtracking que no muestra el progreso de la búsqueda obtiene la solución en unos pocos segundos, aún cuando el tablero requiere un número importante de iteraciones (*Near Worst Case, Con 17 pistas,* etc.). La optimización de este algoritmo se logró utilizando estructuras de datos básicas como las matrices (matriz Casillas, de enteros, y matriz EsPista, de booleanos), en las cuales el acceso a las posiciones que ocupan los valores supone un tiempo constante c_i mínimo para el hardware de hoy día. El tiempo promedio de duración de la resolución de un tablero utilizando el algoritmo de backtracking sin muestra es de 2.9 segundos[9]. La moda, sin embargo, es de 0 segundos (valor que se utiliza para representar tiempos instantáneos), con lo

[9] La media aritmética no es útil como medida de tendencia central en este caso, debido a la existencia de valores extremos. Para su cálculo se utilizaron, en lugar de los tiempos nulos 00:00:00:00, tiempos mínimos de 0.1 segundos.

cual puede afirmarse que el algoritmo trabaja muy eficientemente. Esto puede analizarse en detalle en *Gráfico 1*.

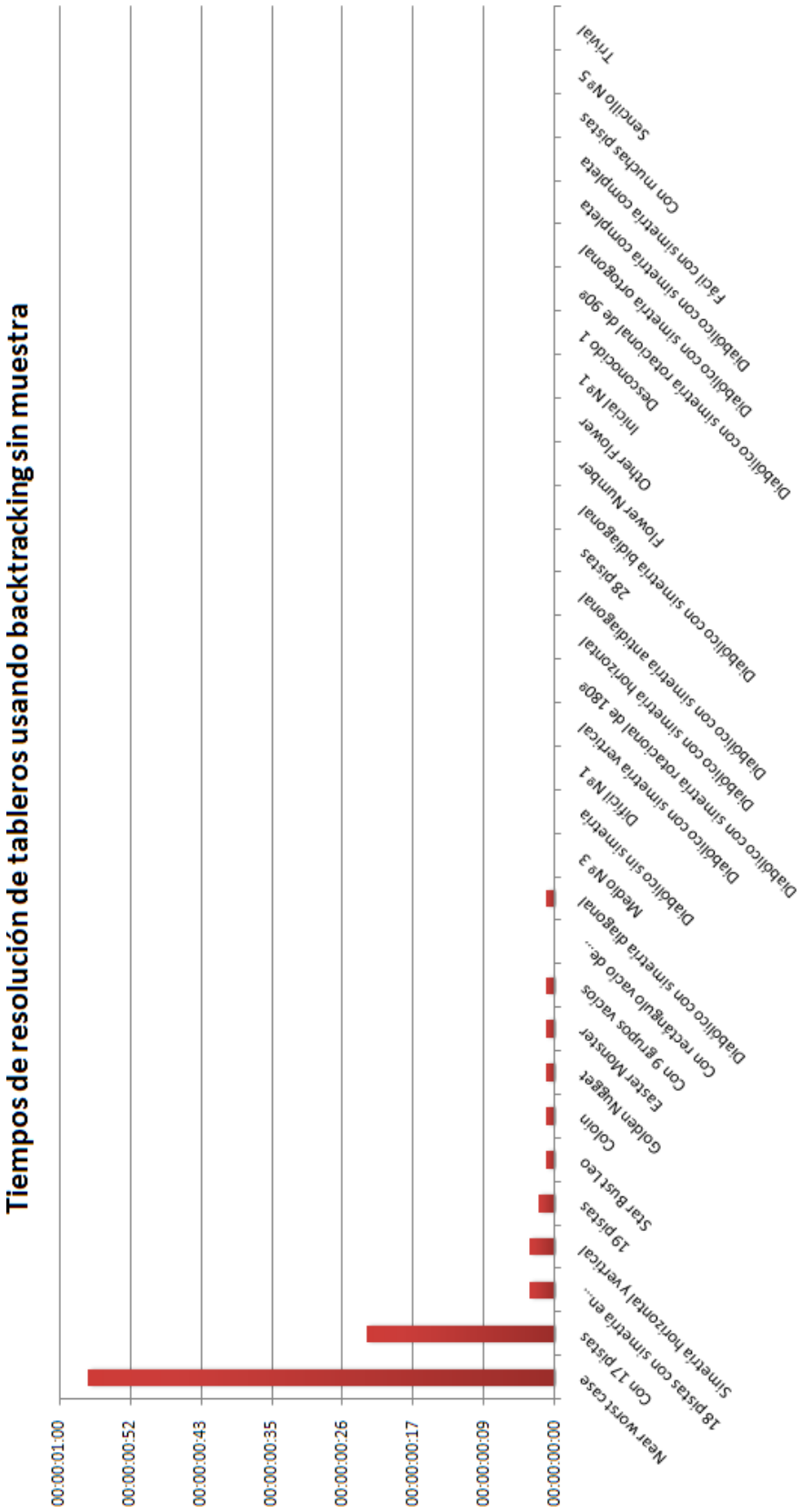

Gráfico 1: *Tiempo de resolución de tableros usando backtracking sin muestra.*

Respecto del algoritmo de backtracking con muestra del progreso, la eficiencia es menor debido al incremento de los tiempos constantes que se necesitan para colocar información en los botones y el acceso a los datos que estos contienen. Esto se observa con mayor claridad en el *Gráfico 2*.

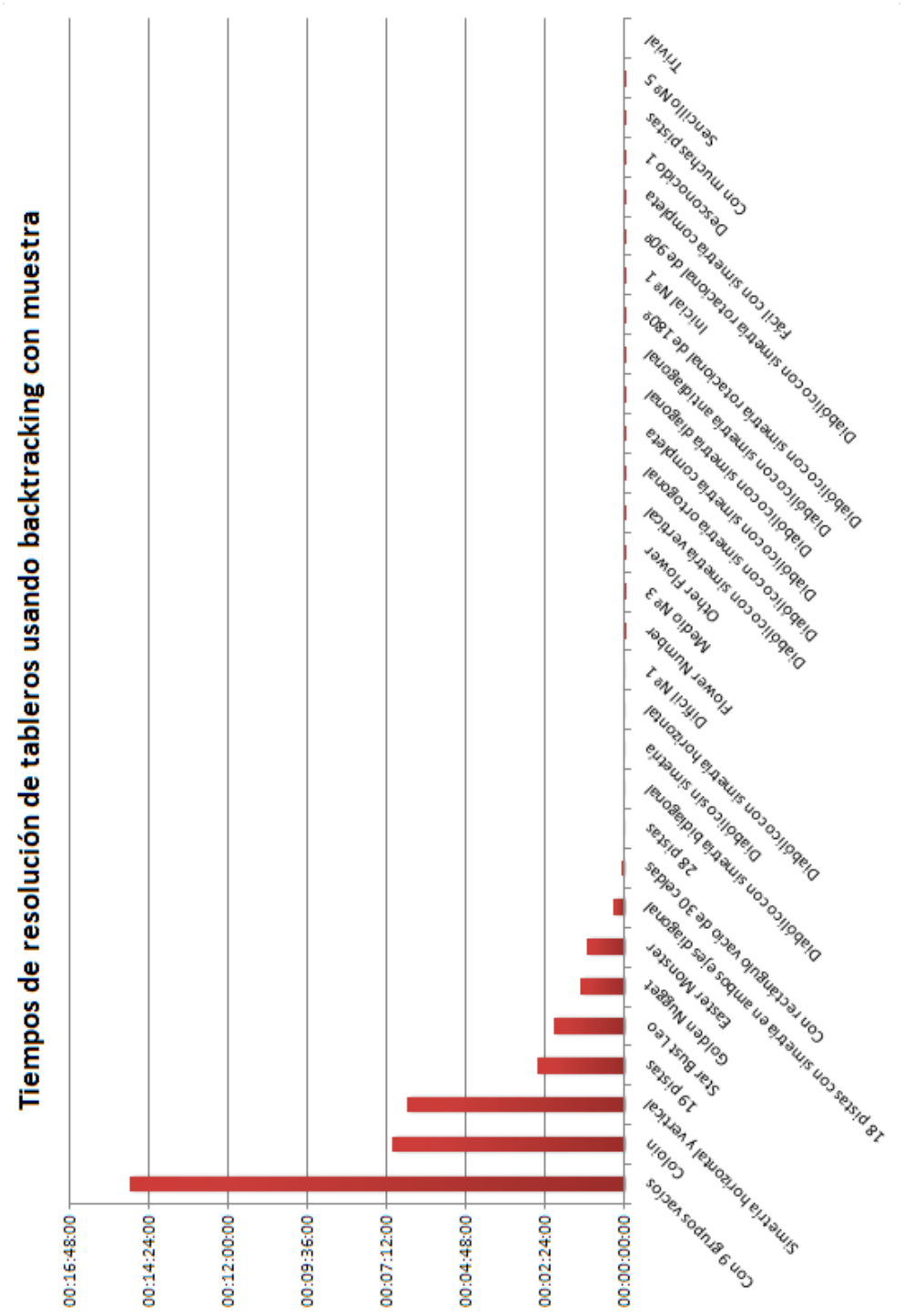

Gráfico 2: *Tiempos de resolución de tableros usando backtracking con muestra. Se omitieron los tableros de mayor demora puesto que afectaban la visibilidad del resto.*

De acuerdo a relaciones de proporcionalidad realizadas, la cantidad de iteraciones por segundo (valor estimado promedio) que requiere este algoritmo es de 566. Si tenemos en cuenta que, realizando cálculos similares, la cantidad de iteraciones por segundo que ejecuta el algoritmo sin muestra es de $10.663.650$ [10], está claro que el incremento de los tiempos constantes en el algoritmo con muestra es muy significativo. Si llamamos c_s a la constante del algoritmo sin muestra y c_m a la constante del otro algoritmo, podemos aproximar $c_m = 18840 \times c_s$.

El tiempo promedio de demora de resolución de un tablero utilizando el algoritmo con muestra es de 1 hora, 13 minutos y 9 segundos, aunque es un valor poco representativo teniendo en cuenta valores extremos como los del tablero *Near Worst Case* o *Con 17 pistas*. La moda (2 segundos) tampoco es representativa en su totalidad, puesto que el valor de repetición es ínfimo (3) y que no es una cifra importante dada la dimensión acotada de la población.

5.3. Conclusiones

A partir de los estudios realizados en las secciones 5.1 y 5.2, podemos concluir que el más eficiente de los dos algoritmos es el de backtracking sin muestra, a pesar de que ambas complejidades temporales son iguales. Aunque éste carezca del atractivo visual del otro algoritmo, ofrece una respuesta tan inmediata que el usuario deberá sacrificarse tan sólo en ciertos casos especiales, de una mayor duración. Se ha dejado en el programa final para que sea el usuario el que decida cuál de los dos algoritmos desea utilizar para resolver sus tableros, y para realizar los posteriores estudios de eficiencia, que ejemplifican con claridad la importancia que poseen los tiempos constantes a la hora de lograr un algoritmo voraz de alto rendimiento.

Por último, podemos afirmar que en la búsqueda de soluciones para tableros sudokus el algoritmo de backtracking es uno de los mejores posicionados respecto a su eficiencia siempre y cuando se implemente utilizando podas y trabajando profundamente la relación de algoritmo/interfaz de tal forma que los tiempos constantes se reduzcan a lo mínimo e indispensable.

[10] El valor es aproximado, teniendo en cuenta la carencia de precisión que acarrea, a la hora de medir tiempos, el uso de un cronómetro manual.

Nombre	Dif.	Pistas	Iteraciones	Densidad de Nº altos en primeras filas
Near worst case	4	17	622.577.597	Alta
Con 17 pistas	4	17	239.312.393	Alta
Con 9 grupos vacíos	3	22	30.579.053	Media
Coloin	4	21	14.366.573	Media
Simetría horizontal y vertical	2	19	13.494.200	Alta
19 pistas	3	19	5.415.898	Alta
Star Bust Leo	4	20	4.355.517	Alta
Golden Nugget	4	21	2.736.230	Media
Easter Monster	4	21	2.357.880	Baja
18 pistas con simetría en…	3	18	672.403	Baja
Con rectángulo vacío de…	3	22	183.021	Media
28 pistas	1	28	120.517	Media
Diabólico con simetría bidiagonal	4	28	110.161	Baja
Diabólico sin simetría	4	25	63.211	Media
Diabólico con simetría horizontal	4	27	52.367	Alta
Difícil Nº 1	4	26	44.387	Baja
Flower Number	1	30	37.652	Baja
Medio Nº 3	3	25	31.066	Baja
Other Flower	1	30	14.096	Media
Diabólico con simetría vertical	4	26	7.389	Media
Diabólico con simetría ortogonal	4	34	5.063	Baja
Diabólico con simetría completa	4	34	4.515	Alta
Diabólico con simetría diagonal	4	24	3.804	Media
Diabólico con simetría antidiagonal	4	27	3.156	Baja
Diabólico con simetría rotacional de 180º	4	26	2.364	Baja
Inicial Nº 1	1	31	1.720	Baja
Diabólico con simetría rotacional de 90º	4	33	1.206	Baja
Fácil con simetría completa	1	39	1.089	Media
Desconocido 1	1	33	1.052	Baja
Con muchas pistas	1	40	845	Baja
Sencillo Nº 5	1	40	644	Baja
Trivial	1	56	142	Baja

Tabla 1: *Análisis de eficiencia realizado sobre 32 tableros elegidos al azar de la población de 100 provista con la aplicación, a partir del número de iteraciones necesarias para resolverlos.*

Nombre	Dif.	Pistas	Tiempo con muestra	Tiempo sin muestra	Iteraciones
Near worst case	4	17	12:17:00:20	00:00:00:57	622.577.597
Con 17 pistas	4	17	4:21:14:27	00:00:00:23	239.312.393
Con 9 grupos vacíos	3	22	00:14:58:51	00:00:00:03	30.579.053
Coloin	4	21	00:07:02:17	00:00:00:03	14.366.573
Simetría horizontal y vertical	2	19	00:06:36:39	00:00:00:02	13.494.200
19 pistas	3	19	00:02:39:00	00:00:00:01	5.415.898
Star Bust Leo	4	20	00:02:08:01	00:00:00:01	4.355.517
Golden Nugget	4	21	00:01:20:25	00:00:00:01	2.736.230
Easter Monster	4	21	00:01:09:18	00:00:00:01	2.357.880
18 pistas con simetría en…	3	18	00:00:19:46	00:00:00:01	672.403
Con rectángulo vacío de…	3	22	00:00:05:22	00:00:00:00	183.021
28 pistas	1	28	00:00:03:31	00:00:00:01	120.517
Diabólico con simetría bidiagonal	4	28	00:00:03:14	00:00:00:00	110.161
Diabólico sin simetría	4	25	00:00:01:50	00:00:00:00	63.211
Diabólico con simetría horizontal	4	27	00:00:01:32	00:00:00:00	52.367
Difícil Nº 1	4	26	00:00:01:18	00:00:00:00	44.387
Flower Number	1	30	00:00:01:05	00:00:00:00	37.652
Medio Nº 3	3	25	00:00:00:54	00:00:00:00	31.066
Other Flower	1	30	00:00:00:25	00:00:00:00	14.096
Diabólico con simetría vertical	4	26	00:00:00:13	00:00:00:00	7.389
Diabólico con simetría ortogonal	4	34	00:00:00:09	00:00:00:00	5.063
Diabólico con simetría completa	4	34	00:00:00:08	00:00:00:00	4.515
Diabólico con simetría diagonal	4	24	00:00:00:06	00:00:00:00	3.804
Diabólico con simetría antidiagonal	4	27	00:00:00:06	00:00:00:00	3.156
Diabólico con simetría rotacional de 180º	4	26	00:00:00:04	00:00:00:00	2.364
Inicial Nº 1	1	31	00:00:00:03	00:00:00:00	1.720
Diabólico con simetría rotacional de 90º	4	33	00:00:00:02	00:00:00:00	1.206
Fácil con simetría completa	1	39	00:00:00:02	00:00:00:00	1.089
Desconocido 1	1	33	00:00:00:02	00:00:00:00	1.052
Con muchas pistas	1	40	00:00:00:02	00:00:00:00	845
Sencillo Nº 5	1	40	00:00:00:01	00:00:00:00	644
Trivial	1	56	00:00:00:00	00:00:00:00	142

Tabla 2: *Estudio de eficiencia realizado sobre 32 tableros sudokus, ordenado por número de iteraciones. Los tiempos indicados como 00:00:00:00 representan la obtención de la solución inmediata. Los tiempos con muestra indicados en negrita han sido estimados a partir de una relación de cantidad de iteraciones por segundo.*

6. Referencias

[1] http://es.wikipedia.org/wiki/Sudoku.
Información disponible sobre algunas cuestiones básicas del sudoku.

[2] http://en.wikipedia.org/wiki/Mathematics_of_Sudoku.
Conceptos matemáticos relacionados al juego del sudoku.

[3] http://www.cristalab.com/files/ejemplos/sudoku/sudoku.zip
Archivos fuente para compilar un Sudoku escrito enteramente en ActionScript 2 que implementa un algoritmo de generación y resolución de sudokus.

[4] http://www.cristalab.com/tutoriales/
En el apartado "Algoritmo para generar y resolver sudokus con código Actionscript 2" hay datos referidos a las propiedades aritméticas de un tablero sudoku.

[5] *Euler, Leonhard.* "On Magic Squares". 1776. Traducción del latín al inglés de Jordan Bell, diciembre de 2004. Disponible en Internet en www.eulerarchive.org.
Contiene un detallado análisis matemático de los cuadrados mágicos.

[6] http://es.wikipedia.org/wiki/Sudoku
Información (en español) sobre el sudoku. Incluye detalles muy interesantes sobre la algoritmia y la historia del juego.

[7] *Valverde Rebaza, Jorge Carlos.* "SuDoku: un problema NP-completo". Escuela de Informática, Universidad Nacional de Trujillo, Perú. 2006.
Trabajo de investigación donde se demuestra a través de FNC (forma normal conjuntiva) que la resolución de un sudoku es un problema NP-completo.

[8] http://en.wikipedia.org/wiki/Algorithmics_of_Sudoku.
Disponibles varias ideas para escribir algoritmos para resolver sudokus.

[9] *Brouwer, Andries E.* "Sudoku Puzzles and How to Solve Them". NAW 5/7. 4 de diciembre de 2006. Disponible en internet en http://homepages.cwi.nl/~aeb/games/sudoku.
Explicación detallada de cómo resolver sudokus mediante lógica encadenada.

[10] *Ruiz, José María*. "Sudokus". Linux-Magazine, 2005. Disponible en Internet en http://www.linux-magazine.es/Magazine/Downloads/13.
Ofrece una explicación detallada de un algoritmo inteligente escrito en Python que resuelve sudokus.

[11] *Bello, Luciano*. "Algoritmos Genéticos para la Resolución de Sudokus". Trabajo para la cátedra "Inteligencia Artificial", dirigido por la profesora María Florencia Pollo.
Un gran estudio de eficiencia realizado a partir de un algoritmo genético diseñado por el alumno.

[12] *Cervigón Rückauer, Carlos*. "Operadores genéticos sobre permutaciones aplicados a la resolución del sudoku". Departamento de Ingeniería del Software e Inteligencia artificial. Universidad Complutense de Madrid. Madrid, España.
Un completo análisis de eficiencia de un algoritmo genético desarrollado para el trabajo de investigación.

[13] ***Carbajal, Paredes Luis*****. "Sudoku: informe final". Trabajo para la cátedra "Inteligencia Artificial y Sistemas Expertos". Universidad San Martín de Porres.**
Un trabajo no muy complejo que explica un algoritmo escrito en LISP para resolver sudokus.

[14] ***Chu, Jonathan*****. "A Sudoku Solver in Java implementing Knuth's Dancing Links Algorithm". Trabajo de investigación y desarrollo para la "Harker Research Symposium". 2006.**
Investigación sobre el algoritmo de Knuth de "Dancing Links" e implementación en Java del mismo.