ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA INFORMÁTICA

Pasatiempo lógico: Puzzle Japonés (Hanjie o nonograma)

Realizado por
JAVIER MARTÍN JIMÉNEZ

Dirigido por
VÍCTOR ÁLVAREZ SOLANO

Departamento
MATEMÁTICA APLICADA 1

Sevilla, Septiembre 2009

Pasatiempo lógico:Puzzle Japonés
(Hanjie o Nonograma)

Índice

Introducción

El tema y motivación principal de este proyecto de fin de carrera es el juego de origen japonés llamado **Nonograma** o **Hanjie**.

Definición

Los **Nonogramas** son un tipo de pasatiempo en el que se usa una rejilla rectangular de tamaño variable (normalmente cuadrada y con dimensión múltiple de 5). El objetivo del juego es descubrir el dibujo oculto en la rejilla pintando de negro los cuadros adecuados, completando el tablero a partir de los datos que se dan: cada número de cada fila y columna indica una sucesión de cuadros negros consecutivos en esa fila, separados de otros cuadros negros por al menos un espacio blanco a cada lado. Los números de las entradas horizontales y verticales se dan en el mismo orden en el que van los grupos de cuadros negros, y no se omite ninguno.

Por ejemplo, una entrada horizontal de "1 2" como la tercera de la Figura 1, significaría que en esa fila hay bloques de uno y dos celdas en negro, en ese orden, con al menos un cuadrado en blanco entre los grupos sucesivos.

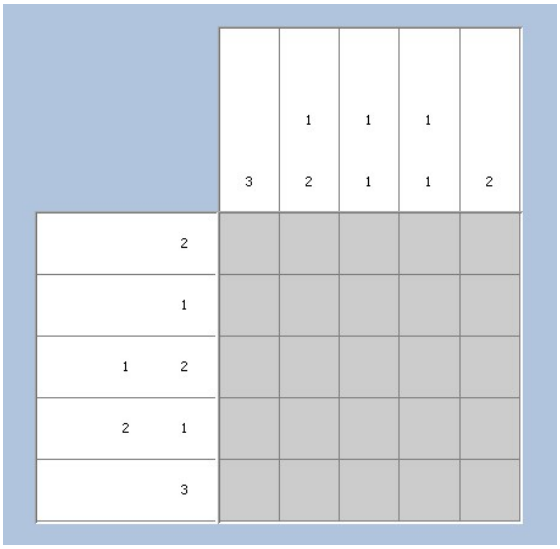


Fig. 1 – Nonograma 5x5 sin resolver

Algunos puzzles de este tipo podrán tener más de una solución, dependiendo de las entradas por filas y columnas del tablero, aunque, en estos casos, generalmente la solución buscada será la que conforme una figura o imagen concreta.

Estos puzzles son normalmente blanco y negro, pero también pueden basarse en varios colores de tal forma que tanto las entradas horizontales como las verticales también serán de un color para indicar el color de los bloque de cuadrados que definen. Dos números de diferente color pueden o no tener un espacio entre ellas. Por ejemplo, una entrada de una fila con un "cuatro negro" seguido de un "dos rojo" podría significar que la fila tiene cuatro celdas en negro, algunos espacios vacíos y dos celdas de color rojo, o simplemente puede significar cuatro casillas en negro seguido de inmediato por dos rojos.

No hay límites teóricos sobre el tamaño de un nonograma y tampoco se limita a diseños rectangulares. Existen nonogramas más complejos con rejillas de seis lados con la utilización de varios colores para componer la imagen solución y otros cuyas celdas son triángulares, denominados *Triddlers*.

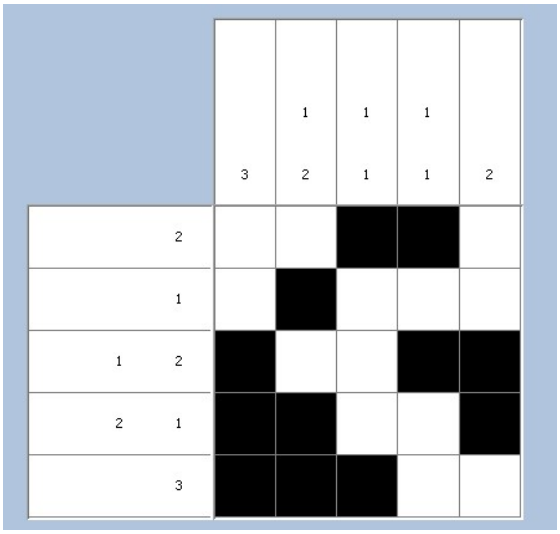


Fig. 2 – Nonograma anterior resuelto.

Historia

Las rejillas con imágenes ocultas han sido usadas en puzzles y pasatiempos desde hace muchos años. Sin embargo, y a pesar de la creencia de que es un juego japonés muy antiguo, los nonogramas son un tipo de puzzle relativamente reciente.

Todo empezó en 1987, cuando la editora gráfica Non Ishida ganó una competición en Tokyo al diseñar imágenes creadas a partir de un rascacielos con luces apagadas y encendidas. Esto la dirigió a idear un pasatiempo basado en el relleno de ciertas casillas de una rejilla. Por otro lado, en el mismo periodo y sin tener ninguna conexión con Non Ishida, un editor de pasatiempos profesional japonés llamado Tetsuya Nishio inventó este mismo tipo de pasatiempo.

En 1988, Non Ishida publicó tres pasatiempos basados en rejillas con imagen en Japón, usando el nombre de "Window Art Puzzles". Estos puzzles empezaron a aparecer en revistas japonesas de pasatiempos.

En 1989, Non Ishida mostró al británico James Dalgety sus pasatiempos. James pensó que la idea de aquellos puzzles era muy buena y llegó a un acuerdo con Non Ishida para comercializar sus diseños fuera de Japón.

En 1990, James Dalgety inventó el nombre de **Nonogram** (Nonograma), obtenido de la combinación de los términos "Non" (de *Non Ishida*) y "gram" (de *diagram*). Más tarde consiguió que el periódico británico *The Sunday Telegraph* publicara estos pasatiempos de forma semanal. James Dalgety, junto a su equipo de diseñadores de puzzle, escribió el que posiblemente sea el primer programa para resolver estos pasatiempos. Actualmente existen programas capaces de resolver nonogramas mucho más complejos; aún así, *The Sunday Telegraph* aún utiliza el algoritmo original para comprobar todos sus puzzles.

En 1993, los nonogramas fueron reimportados por Mainich (uno de los periódicos más importantes de Japón) y se empezaron a publicar regularmente en Japón. Este mismo año apareció el primer libro dedicado a los nonogramas publicado por Non Ishida en Japón. Más tarde, en ese mismo año, sale a luz el libro "*The Sunday Telegraph Book of Nonograms*", publicado en el Reino Unido por la editorial *Pan Books*.

En 1995, aparecieron los primeros nonogramas en juegos electrónicos y videoconsolas portátiles como la famosa *Game Boy* de *Nintendo*, para la que se lanzó el videojuego *Picross*. La popularidad alcanzó tal nivel en Japón que nuevos editores publicarón varias revistas mensuales, algunas de las cuales contenían hasta 100 rompecabezas. La editorial *Pan Books* publicó llegó al cuarto volumen de su libro de nonogramas con "The 4th Sunday Telegraph Book of Nonograms".

También en 1995, Non Ishida quiso usar el nombre "Nonogram" exclusivamente para sus propios diseños en Japón, acabando así con la colaboración de James Dalgety en 1996. James Dalgety es poseedor de los derechos de la marca registrada **Nonogram™** desde 1995.

En 1996, el juego japonés de arcade *Logic Pro* lanzado por la compañía *Deniam Corp*, con secuela del juego liberada el año siguiente.

En 1998, *The Sunday Telegraph* inició una competición para elegir un nuevo nombre para sus puzzles. **Griddlers** fue el nombre ganador que los lectores del periódico eligieron. **Griddlers™** es marca registrada de *Telegraph Group* desde este año.

En 1999, los nonogramas se publicaron en Holanda por *Sanoma Uitgevers*, en Reino Unido por *Puzzler Media* (anteriormente *British European Associated Publishers*) y en Israel por *Nikui Posh Puzzles*.

En 2007, *Nintendo* lanzó al mercado otra versión de *Picross*, en esta ocasión para su consola *Nintendo DS*.

Actualmente, revistas con este tipo de pasatiempo son publicados en los Estados Unidos, Reino Unido, Alemania, Italia, Hungría, Finlandia y muchos otros países. El puzzle es conocido con múltiples nombres, como *Nonograma*, *Hanjie*, *Paint by Numbers*, *Griddlers*, *Pic-a-Pix* o *Picross*.

Descripción y objetivos

El objetivo del presente proyecto es la realización de un programa informático que permita la lectura y generación de nonogramas para que el usuario pueda resolverlos por él mismo, así como la resolución automática de éstos en un tiempo razonable y la posibilidad de encontrar todas las soluciones en el caso de que un puzzle tuviera más de una. Se permitirá "ver" la ejecución del algoritmo que resuelve el puzzle, con la posibilidad de poder avanzar "paso a paso", de modo que el algoritmo se detiene cada vez que se marca una celda de tablero del juego.

El programa aceptará nonogramas en blanco y negro y cuadrados, pudiendo leer y generar tableros de 5x5, 10x10, 15x15 o 20x20. La lectura de los puzzles podrá hacerse mediante la carga de ficheros de textos con una estructura definida, o bien mediante la digitalización de archivos de imágenes (.jpg, .gif, .png, …) para generar tableros con las dimensiones que se indiquen. Estas conversiones se podrán salvar en ficheros de texto para su posterior carga en el programa.

Para la resolución de los puzzles, se realizará un algoritmo basado en algunas técnicas que se explicarán más adelante.

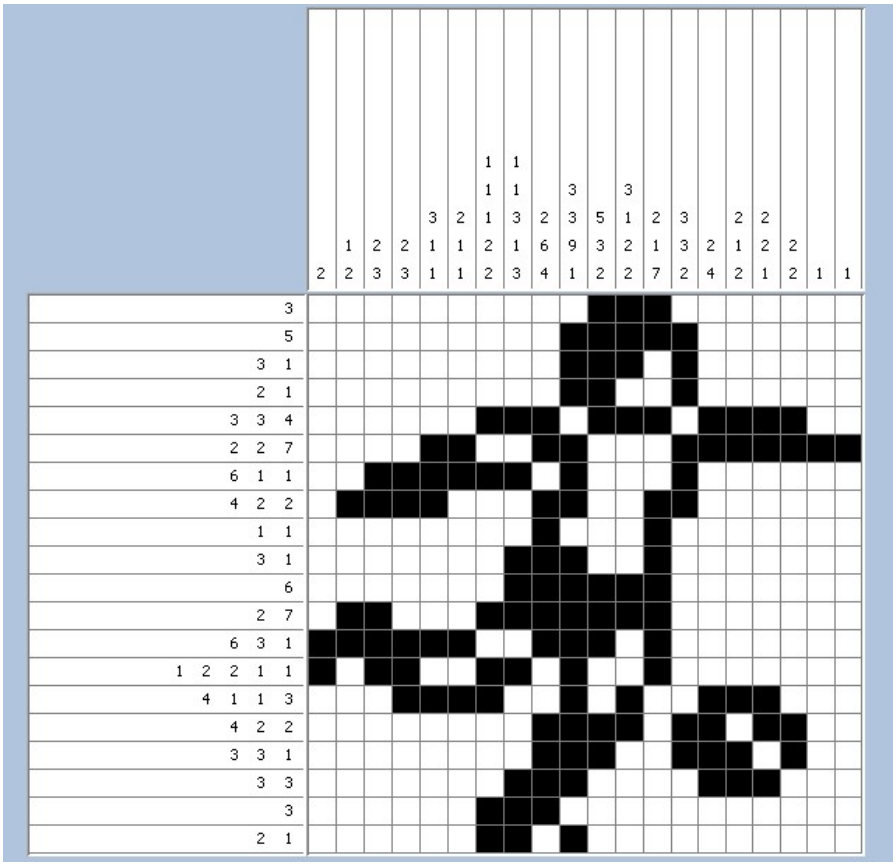


Fig. 3 – Nonograma 20x20 resuelto.

Herramientas utilizadas

El lenguaje de programación utilizado para la implementación de los distintos algoritmos así como de la interfaz gráfica de la aplicación es **Java**. Posiblemente otros lenguajes de programación más específicos y orientados para la resolución de este tipo de problemas como **Prolog** o **Lisp** hubieran requeridos menos líneas de códigos para implementar el algoritmo que resuelve los puzzles, siendo así más eficientes y rápidos que programando con lenguaje Java.

Entonces, ¿por qué utilizar Java?

El motivo principal de usar Java es la integración del código que resuelve los nonogramas con la interfaz gráfica de la aplicación y poder así aumentar la funcionalidad y jugabilidad para el usuario.

Además, al tratarse de un lenguaje orientado a objetos, se facilita el diseño de la estructura de datos necesario para la lectura y generación de los tableros, la propia resolución del tablero y la presentación de resultados.

A todo esto, añadir las ventajas propias del mismo lenguaje (portabilidad, seguridad, aceptación y consagración del lenguaje, información en la web,….).

La máquina virtual de Java (JVM) que se ha usado para el desarrollo de esta aplicación es la versión **Java 1.6.0_06**, por lo que el programa está optimizado para esa versión (descarga gratuita en http://java.sun.com/javase/downloads/index.jsp)

Por otro lado, el entorno de desarrollo elegido para la programación de todo el proyecto es **NetBeans IDE**, concretamente su versión **6.7** (Build 200906241340). La predilección de este software para desarrollar la aplicación no es fortuita, y éstos han sido los factores que han influido para la elección de este entorno:

- Se trata de un entorno de desarrollo gratuito, disponible para todo el mundo en el sitio http://www.netbeans.org.

- Al ser una plataforma gratuita, existe una gran cantidad de información en la web en forma de manuales *online* o en los distintos foros para resolver las posibles dudas o dificultades.

- A diferencia de otros entornos gratuitos, Netbeans IDE incorpora un editor gráfico bastante intuitivo así como librerías propias de Netbeans muy útiles a la hora de realizar la interfaz gráfica sin necesidad de instalar *plugins* adicionales.

Este último punto puede llegar a ser un inconveniente: al utilizarse librerías no estándares de la versión de Java que se use, se crea dependencia de este entorno. Esto se solucionaría con la exportación/importación de dichas librerías al entorno de desarrollo que se quisiera utilizar.

Desarrollo

Por encima de todas las funcionalidades de la aplicación a desarrollar, la más importante y atractiva es la **resolución automática** de los nonogramas que acepta el programa.

Resolución del problema

Resolver puzzles nonogramas es un problema **NP-completo**, lo que significa que no existe ningún algoritmo de tiempo polinómico (tiempo razonable) que resuelva todos los puzzles que se generen.

En la teoría de la complejidad computacional, la clase de complejidad NP-completo es una clase de problemas que tienen dos propiedades:

- Cualquier solución dada al problema se puede verificar rápidamente (en tiempo polinomial). El conjunto de problemas con esta propiedad se llama *NP.*
- Si un problema *C* perteneciente a *NP* se puede resolver rápidamente (en tiempo polinomial), entonces cada problema en *NP* también lo hará.

La característica más notable de los problemas NP-completos es que no se conoce la solución rápida a ellos. Es decir, el tiempo necesario para resolver el problema con los algoritmos que actualmente se conocen aumenta en cuanto crece el tamaño del problema a resolver. Actualmente, determinar si es o no es posible resolver estos problemas con rapidez es uno de los principales problemas por resolver por la teoría de la computación.

Los problemas NP-completos se abordan mediante el uso de algoritmos de aproximación.

Transportando todo esto al problema de resolver nonogramas, se podría decir que es fácil **verificar** si un tablero concreto es solución o no del puzzle (para todo puzzle), pero no existe ningún algoritmo que permita **encontrar** la solución de todos los puzzles que se aborden.

Esta complejidad computacional no será un problema para los puzzles que se pretenden resolver con la aplicación. Los tamaños que se admiten en ésta son 5x5, 10x10, 15x15 y 20x20, de modo que son razonablemente solucionables por cualquier persona. Aún así, se pretende que el programa pueda solucionar cualquier puzzle que admita, y para ello se utilizarán algoritmos de aproximación.

Existen varias técnicas que ayudan a resolver nonogramas. A continuación se describirá la que se ha utilizado para este proyecto.

Para resolver un nonograma, es necesario determinar qué celdas serán negras y cuáles en blanco. La determinación de las celdas blancas es tan importante como la determinación de las celdas negras. Existen dos técnicas concretas para intentar determinar tanto celdas negras como blancas, y muy útiles para iniciar la búsqueda de una solución.

Simple Boxes

Este método usa las conjunciones de los posibles lugares que puedan ocupar los bloques negros en una fila o columna. Por ejemplo, si en un tablero 10x10 hay una fila que tiene una entrada con "4 3", calculando todas las combinaciones posibles de los bloques de 4 y 3 celdas negras, se tiene que todas las filas posibles tienen la tercera, cuarta y octava celda marcada en negro, por lo que forzosamente la solución del tablero tendrá dichas celdas marcadas en negro.

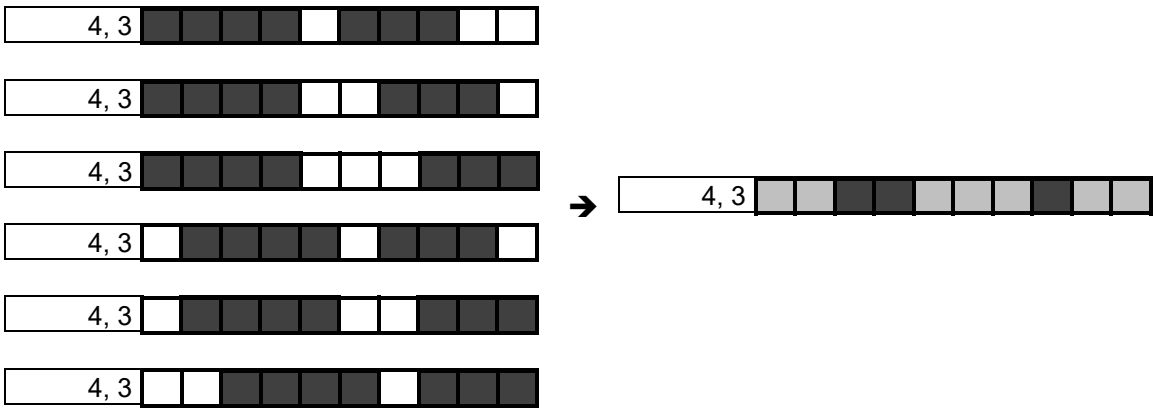


Fig. 4– Simple Boxes.

Simple Spaces

Este método es análogo al anterior, pero en este caso se usa para determinar celdas en blanco. Para que este método sea efectivo, se debe aplicar sobre filas y columnas que ya tengan algunas celdas marcadas en negro o en blanco (de otra forma nunca marcaría ninguna casilla en blanco). Es decir, este método determina los espacios mediante la búsqueda de celdas que están fuera del alcance de cualquier posible bloque de celdas negras. Por ejemplo, para una fila que tiene "3 1" como entrada, y la fila ya tiene marcadas las celdas 4 y 9, se calculan las posibles distribuciones de los bloques de celdas negras, quedando sin marcar la primera, séptima, octava y décima casilla sin marcar en todas las combinaciones.

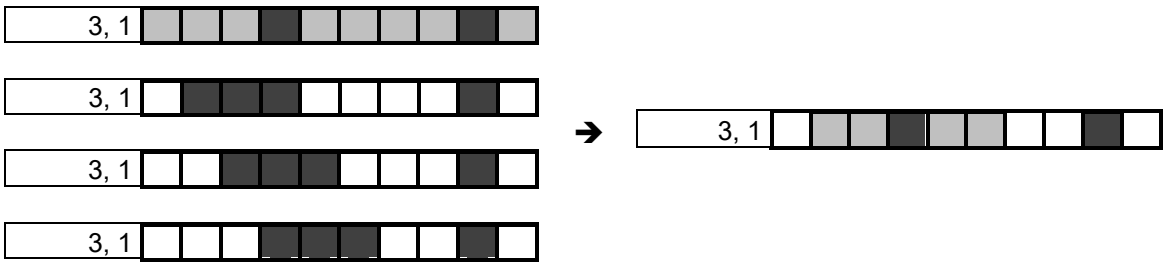


Fig. 5 – Simple Spaces.

Como se puede observar, con la determinación de celdas negras se facilita la determinación de celdas blancas, y viceversa.

El algoritmo utilizado en este proyecto se inicia con una función que combina de ambas técnicas, método que he llamado ***SimpleBoxesSpaces***, en el que se aplican los métodos *Simple Boxes* y *SimpleSpaces*, primero por filas y después por columnas. Esto se realiza repetidamente mientras se vayan descubriendo nuevas celdas del tablero solución. Los parámetros de entrada de esta función son el tamaño del Juego, un array con las entradas por filas (horizontales) del juego, un array con las entradas por columnas (verticales) y un objeto que representa el tablero actual del puzzle.

```
SimpleBoxesSpaces(entero tamañoJuego, entero [] entradasH, entero [] entradasV,
tableroHanjie tablero)

    booleano hayCambios = verdadero;

    mientras hayCambios

      hayCambios = falso;
      entero i = 0;
      mientras i< tamañoJuego
        hayCambios = SimpleBoxesSpacesPorFilas(i,entradasH[i], tamañoJuego, tablero) O
        hayCambios;
        i = i + 1;
      fmientras

      entero j = 0;
      mientras j< tamañoJuego
        hayCambios = SimpleBoxesSpacesPorColumnas(i,entradasV[j], tamañoJuego,
        tablero) O hayCambios;
        j = j + 1;
      fmientras

    fmientras

fin
```

Las funciones *SimpleBoxesSpacesPorFilas* y *SimpleBoxesSpacesPorColumnas* implementan los métodos SimpleBoxes y SimpleSpaces por filas y por columnas, respectivamente.

El método **SimpleBoxesSpacesPorFilas** tiene como argumento de entrada la entrada horizontal "i" del juego, y determina en primer lugar todas las combinaciones posibles que son factibles con esa entrada "i", de la misma forma que se muestra en las figuras 4 y 5. Después, para cada combinación obtenida se comprueba si el estado actual del tablero puede aceptar esa combinación, descartándose ésta en caso negativo. Mediante las combinaciones que pasan esta criba, se detectan aquellas celdas que son comunes en todas estas combinaciones. Por ejemplo, si la tercera celda es blanca y la quinta celda es negra para todas las combinaciones que quedan, significará que la fila "i" del tablero solución tendrá la tercera celda marcada en blanco y la quinta casilla en negro.

Devuelve “verdadero” si se marca alguna celda del tablero, es decir, si se cambia el estado del tablero.

La función **SimpleBoxesSpacesPorColumnas** es análoga a *SimpleBoxesSpacesPorFilas*, pero en este caso relativo a las columnas y a las entradas verticales del puzzle.

El algoritmo **SimpleBoxesSpaces** termina cuando no se encuentren más celdas nuevas, ya sea porque se haya encontrado la solución o bien porque no se puedan determinar más celdas (blancas o negras) mediante este método. Si se produce lo primero, se devuelve el tablero resuelto y se muestra el resultado obtenido terminando así la resolución del problema; pero, ¿qué hacer si ya no podemos determinar más celdas?

La solución propuesta en este caso se basa en la **búsqueda de contradicciones por recursión**. Esta técnica consiste en detectar una celda cuyo estado aún no se haya determinado y marcarla en blanco (o bien en negro). Entonces, se vuelve a aplicar el algoritmo anterior (*SimpleBoxesSpaces*) para obtener más casillas de la solución. Después de cada iteración del algoritmo *SimpleBoxesSpaces*, se comprueba si el estado del puzzle es correcto: si el tablero es correcto, se vuelve a repetir el proceso de forma recursiva, buscando una nueva celda sin marcar y repitiendo el proceso; en el caso de que el tablero no sea correcto, se habrá llegado a una contradicción, y se desechará el camino recorrido con este método para repetir el proceso marcando la celda anterior en negro, si antes se marcó en blanco, o en blanco, se si probó definiendo la casilla en negro. Si marcando la celda con el color opuesto al que se eligió en el primer intento se vuelve a llegar a una contradicción significará que la opción marcada en un paso anterior del proceso recursivo es incorrecta, por lo que habrá que deshacer todo el camino y probar con el color opuesto de la celda elegida en ese paso recursivo si procede. En el caso de que esto último (se llegue a una contradicción tanto marcando una celda en blanco como en negro) ocurra en el primer paso del proceso recursivo (profundidad cero del algoritmo) significará que el tablero no tiene solución.

De esta forma, el algoritmo que resuelve los nonogramas tendría una estructura similar a la siguiente:

```
Solucion(entero tamañoJuego, entero [] entradasH, entero [] entradasV,
tableroHanjie tablero)

   SimpleBoxesSpaces(tamañoJuego, entradasH, entradasV, tablero);

   si no esSolucion(tablero) entonces
      tablero = solucionRecursiva(tablero, entradasH, entradasV);
      si esSolucion(tablero) entonces
            escribir “Solución encontrada”;
      otras:
         escribir “Nonograma sin solución”;
      fsi
   otras:
      escribir “Solución encontrada”;
   fsi
fin
```

Teniendo en cuenta la estructura anterior, el algoritmo recursivo **solucionRecursiva** tendría la siguiente definición:

```
TableroHanjie solucionRecursiva(TableroHanjie tablero, int [][]
entradasH, int [][] entradasV)

  TableroHanjie tableroAux = nulo;
  entero tamañoJuego = tablero.getTamaño();

    si esSolucion(tablero) entonces
      return tablero;
    otras:
      booleano cambioDeColor = falso;
      //se copia el tablero de entrada:
      tableroAux = new TableroHanjie(tablero);
      //marca la primera casilla que no tenga estado en Blanco
      buscaYMarca(tableroAux, "BLANCO");
      simpleBoxesSpaces(tableroAux, tamañoJuego, entradasH, entradasV);

      si tableroAux.isCorrecto() entonces
        tableroAux=solucionRecursiva(tableroAux, entradasH, entradasV);
           si !tableroAux.isCorrecto() entonces
                cambioDeColor = verdadero;
           fsi
      otras:
        cambioDeColor = verdadero;
      fsi

      si cambioDeColor == verdadero entonces
        //deshago el camino recorrido
        //marcando "la casilla" sin estado:
        tableroAux = new TableroHanjie(tablero);
        //… y la marca ahora en negro
        buscaYMarca(tableroAux,"NEGRO");
        simpleBoxesSpaces(tableroAux, tamañoJuego, entradasH,
entradasV);

        si tableroAux.isCorrecto() entonces
           tableroAux = solucionRecursiva(tableroAux, entradasH,
entradasV);
        fsi
      fsi
    fsi

    return tableroAux;

fin
```

La función **buscaYMarca** selecciona la primera celda que no tenga estado (no sea ni negra ni blanca) y la marca con en blanco o negro, según se indique con el segundo parámetro de entrada.

La función **isCorrecto()** indica si el estado actual del puzzle es correcto o no, es decir, si se encuentran o no contradicciones contrastando el tablero con las entradas horizontales y verticales del nonograma.

La función **esSolucion()** comprueba si el puzzle está resuelto. Esta comprobación la realiza utilizando la función anterior (*isCorrecto()*) y viendo si el número de celdas sin determinar es cero o no.

El funcionamiento del algoritmo se entenderá mejor con un ejemplo. En la siguiente figura se puede apreciar los primeros pasos resultantes de aplicar el algoritmo **SimpleBoxesSpaces** al nonograma de la *figura 1*.

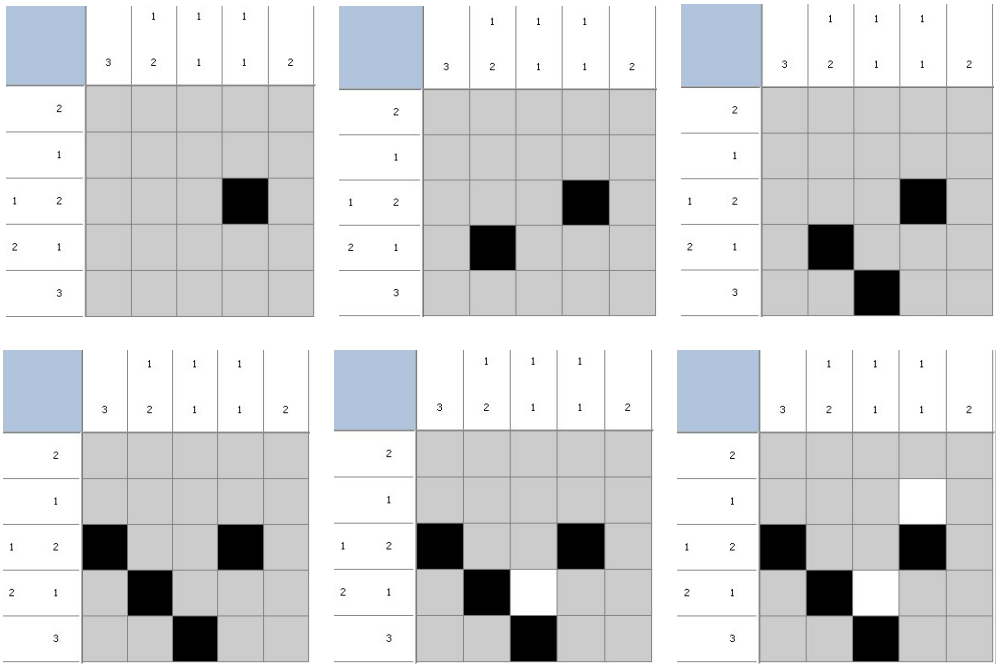


Fig.6 – Primeros pasos de resolución mediente SimpleBoxesSpaces.

El método SimpleBoxesSpaces no resolvería en esta ocasión el puzzle, y devolvería el tablero como se puede ver en la *figura 7.*

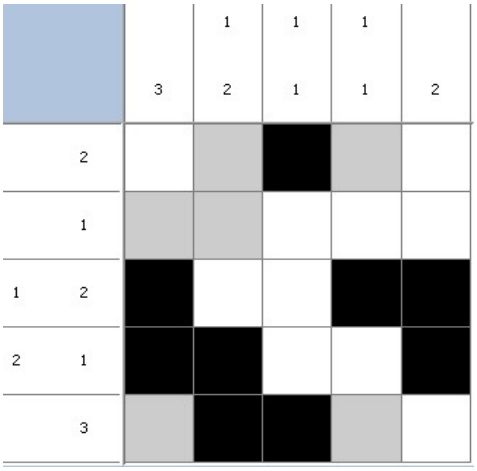


Fig. 7 – Tablero que devuelve SimpleBoxesSpaces aplicado al nonograma de la figura 1.

A partir de este estado del puzzle, se aplicaría el algoritmo recursivo para buscar una solución al tablero. Como se explicó anteriormente, se marcaría la primera celda que se encuentre sin estado. En este caso, sería la segunda celda de la primera fila, pues la búsqueda para marcar se realiza por filas. Se marca esta celda como blanca. Cuando se entra en la función recursiva se cambian los colores de las celdas: las celdas negras pasan a ser naranjas y las blancas, celestes. La celda recién marcada como blanca aparece ya como celeste. Esto es así para comprobar en tiempo de ejecución, mientras se busca la solución, que se está "probando" y recalculando el tablero y, en muchas ocasiones, dibujando tableros incorrectos.

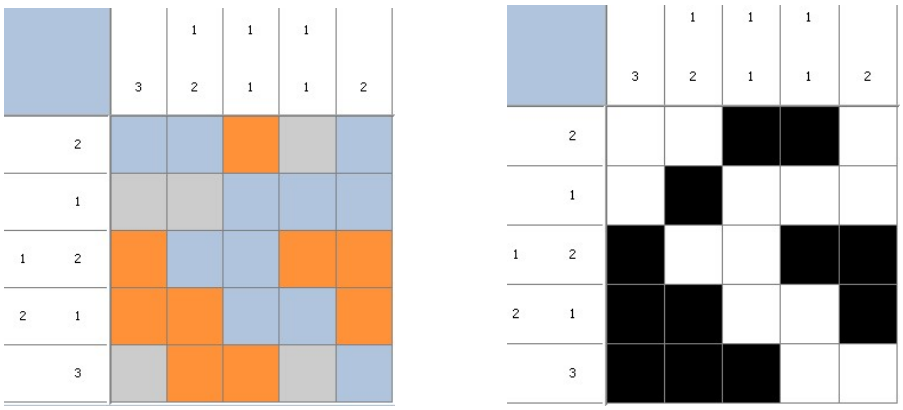


Fig. 8 – En la izquierda, estado del tablero tras marcar la primera celda sin estado que se encuentra. El tablero de la derecha muestra la solución del puzzle, obtenida a partir de aplicar el método SimpleBoxesSpaces al tablero de la izquierda en el primer paso recursivo.

Una vez marcada la celda como blanca (celeste), se vuelve a ejecutar el método *SimpleBoxesSpaces*, llegándose ahora a la solución del nonograma (*Fig. 8*). En este caso se ha llegado a la solución en el primer paso iterativo (profundidad 1).

Si se quieren obtener todas las soluciones de un tablero que no tenga solución única, los cambios a realizar en el algoritmo anterior son mínimos:

```
TableroHanjie solucionRecursiva(TableroHanjie tablero, int [][]
entradasH, int [][] entradasV, Vector soluciones, booleano
buscarTodasSoluciones)

  TableroHanjie tableroAux = nulo;
  entero tamañoJuego = tablero.getTamaño();

   si esSolucion(tablero) entonces
        soluciones.add(tablero);
        return tablero;
   otras:
        booleano cambioDeColor = falso;
        tableroAux = new TableroHanjie(tablero); //se copia el tablero
de entrada
        //marca la primera casilla que no tenga estado en Blanco
        buscaYMarca(tableroAux, "BLANCO");
        simpleBoxesSpaces(tableroAux,      tamañoJuego,      entradasH,
entradasV);
        si tableroAux.isCorrecto() entonces
            tableroAux   =   solucionRecursiva(tableroAux,   entradasH,
entradasV, soluciones, verdadero);

            si buscarTodasSoluciones == verdadero entonces
                cambioDeColor = verdadero;
            otras:
                si !tableroAux.isCorrecto() entonces
                    cambioDeColor = verdadero;
                fsi
            fsi

        otras:
            cambioDeColor = verdadero;
        fsi

        si cambioDeColor == verdadero entonces
      //deshago el camino recorrido marcando "la casilla" sin estado:
           tableroAux = new TableroHanjie(tablero);
           buscaYMarca(tableroAux,"NEGRO");     //…y la marca ahora en
negro
           simpleBoxesSpaces(tableroAux,    tamañoJuego,    entradasH,
entradasV);

           si tableroAux.isCorrecto() entonces
               tableroAux  =  solucionRecursiva(tableroAux,  entradasH,
entradasV, soluciones, verdadero);
           fsi
        fsi
   fsi

   return tableroAux;

fin
```

En el vector de entrada *soluciones*, que se comporta como un parámetro de entrada y salida, se van acumulando todas las soluciones que pueda tener el nonograma.

El nonograma de la *figura 1* posee dos soluciones. Aplicando el algoritmo modificado para obtener todas las soluciones del puzzle, se llega a la solución vista anteriormente además de la que se muestra en la *figura 9*:

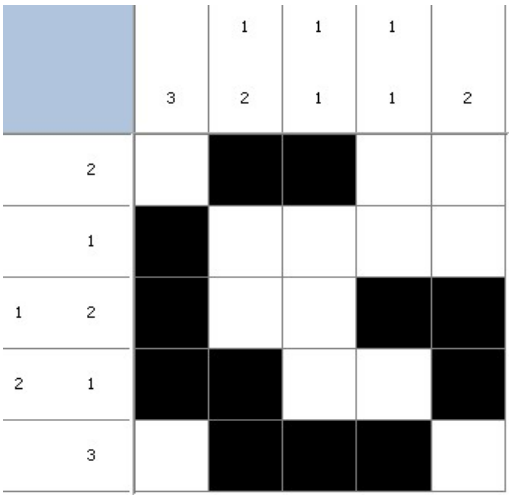


Fig. 8 – Segunda solución al nonograma de la figura 1.

Teniendo claro el funcionamiento del algoritmo utilizado para resolver los nonogramas se puede hacer una idea de que el tiempo de resolución crecería si aumentáramos las dimensiones del pasatiempo.

Por otro lado, también se puede observar lo fácil que sería aumentar la funcionalidad de las funciones que resuelven el problema, así como integrar el algoritmo con una interfaz gráfica, para que los pasos de éste se puedan reflejar en él y se muestren resultados de forma atractiva para el usuario.

(Nota: la descripción detallada del funcionamiento de estas y otras funciones se puede ver en el apartado *Anexo B: Código fuente* de este documento.)

Convertidor de imágenes

Otra de las funcionalidades a destacar es el **digitalizador de imágenes** que incorpora la aplicación. Existen dos formas de cargar un tablero para iniciar el pasatiempo. La primera es leyendo un fichero de texto con un formato específico (que se verá más adelante). La segunda es leyendo un archivo de imagen de cualquier formato (png, jpg, gif,…) y convirtiendo esa imagen en una malla de celdas blancas y negras.

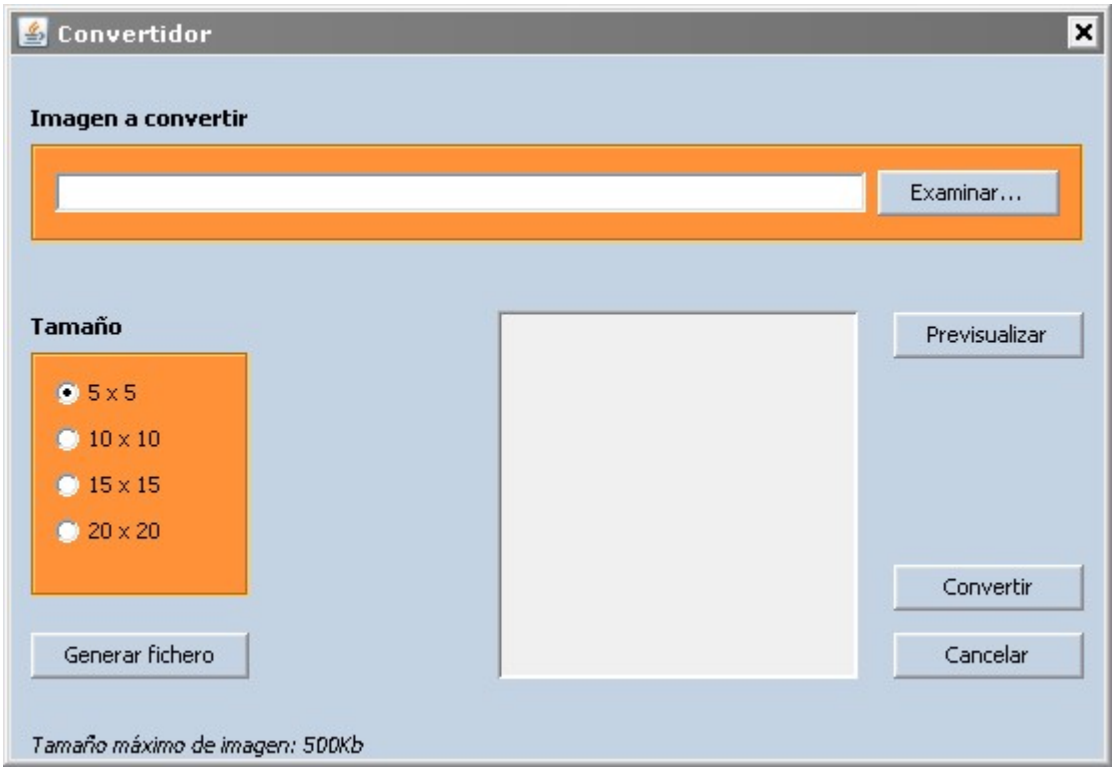


Fig. 9 – Interfaz gráfica del Digitalizador de imágenes.

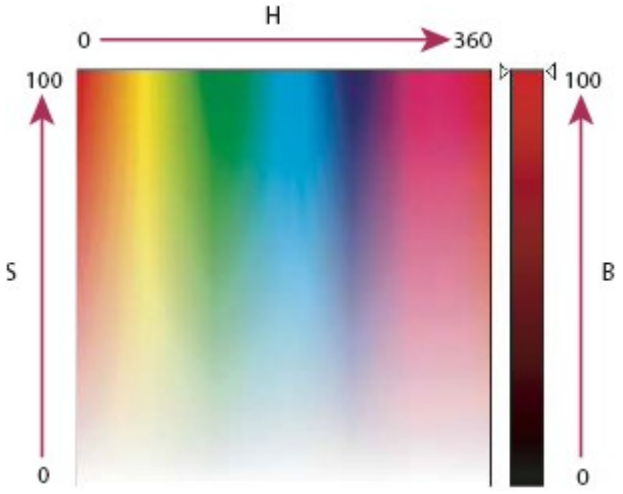
Como se puede apreciar en la figura 5, el funcionamiento del digitalizador es bastante básico y sencillo. Los únicos parámetros necesarios para realizar la conversión son el archivo de imagen y el tamaño del puzzle que se quiere generar. Además, se añade la opción de poder generar un fichero de texto de la imagen y tamaño seleccionados que contendrá la información y el formato necesario para que la aplicación principal pueda posteriormente leerlo y generar el nonograma a partir de este fichero.

Por motivos de eficiencia, se limita el tamaño de la imagen a convertir a medio megabyte, 500 Kb.

El algoritmo para digitalizar las imágenes en una rejilla de celdas blancas y negras se basa principalmente en el modelo de color HSV.

El modelo **HSV** (del inglés *Hue, Saturation, Value* – Tonalidad, Saturación, Valor), también llamado **HSB** (*Hue, Saturation, Brightness* – Tonalidad, Saturación, Brillo), define un modelo de color en términos de sus componentes constituyentes en coordenadas cilíndricas:

- Tonalidad: el tipo de color (como rojo, azul o amarillo). Se representa como un grado de ángulo cuyos valores posibles van de 0 a 360° (aunque para algunas aplicaciones se normalizan del 0 al 100%). Cada valor corresponde a un color. Ejemplos: 0 es rojo, 60 es amarillo y 120 es verde.
- Saturación: Se representa como la distancia al eje de brillo negro-blanco. Los valores posibles van del 0 al 100. Cuanto menor sea la saturación de un color, mayor tonalidad grisácea habrá y más decolorado estará.
- Brillo: Representa la altura en el eje blanco-negro. Los valores posibles van del 0 al 100%. 0 siempre es negro. Dependiendo de la saturación, 100 podría ser blanco o un color más o menos saturado.



Modelo de color HSB

H. Tono

S. Saturación

B. Brillo

Fig. 10 – Gráfico del modelo del color HSB

Por lo tanto, la **saturación** y el **brillo** serán los componentes en el que se centrará el algoritmo para realizar la conversión a una rejilla compuesta de casillas blancas y negras.

La idea básica del convertidor es leer pixel a pixel la imagen seleccionada. De cada pixel, se obtiene su color, y de este color se obtienen los componentes RGB. Con las herramientas Java disponibles, se transforma el valor RGB del color leído en un valor HSB. Con este valor, tenemos las tres componentes que definen el modelo HSB.

Tras leer los valores de saturación y brillo de cada pixel, calculamos la media de estos valores en la imagen y creamos una malla, con la misma dimensión de la imagen original, que contendrá ceros y unos. Una celda de esta malla tendrá un cero si el pixel correspondiente a la

posición de la misma tiene mayor brillo que la media de brillo calculada y además tiene menor saturación que la media de saturación. En caso contrario, esa celda contendrá un uno. A partir de esta malla de ceros y unos se obtendrá el nonograma resultante. Para ello, se realizará un redimensionamiento de la malla, transformando ésta en otra con las dimensiones indicadas (5x5, 10x10, 15x15 o 20x20).

La malla de ceros y unos se divide en bloques cuadrados. La dimensión de estos pequeños subgrupos de ceros y unos dependerá del tamaño de la imagen y del tamaño de entrada. El ancho de estos nuevos subgrupos será igual al ancho de la imagen original dividido entre el tamaño de entrada. El alto de estos subgrupos se calcula de forma análoga.

```
entero bloqueAncho = imagen.getWidth()/tamaño;
entero bloqueAlto = imagen.getHeight()/tamaño;
```

Tendremos así el mismo número de subgrupos que de celdas del tablero final. Una celda del puzzle será de color negro si el subgrupo que le corresponde a esa celda por su posición tiene mayor número de unos que de ceros; en caso contrario, la celda será blanca.

El algoritmo que realiza la digitalización de imágenes quedaría estructurado de esta forma:

```
booleano convertirImagen(entero tamaño, File imagen,
ResultadoConvertidor resultado)

        entero [][] tablero = generarTablero(tamaño, imagen);
        entero [][] entradasH = new entero[tamaño][];
        entero [][] entradasV = new entero[tamaño][];

        si generarEntradas(entradasH, entradasV, tablero) entonces

            resultado.setTablero(tablero);
            resultado.setEntradasH(entradasH);
            resultado.setEntradasV(entradasV);
            resultado.setTamaño(tamaño);
            return verdadero;

        otras:
            return falso;
        fsi
fin
```

donde la función **generarTablero** devolvería el tablero mediante el proceso explicado anteriormente. La función **generarEntradas** se encarga de generar las entradas horizontales y verticales del juego. Esto función no tiene más complejidad que la de leer el tablero recién generado y almacenar los valores en tablas de enteros. El resultado se almacena en un objeto de clase *ResultadoConvertidor*, parámetro de entrada y salida que devuelve el tablero solución y las entradas del juego.

La opción de generar un fichero de texto a partir de una imagen también es trivial. Se ejecuta el anterior algoritmo y con los datos obtenidos y se genera un fichero de texto plano (.txt), con un formato que se verá en el siguiente apartado (Manual de usuario).

Esta forma de transformación hace intuir que la digitalización de la imagen será más eficaz con imágenes cuadradas (más aún si el tamaño es múltiplo de las dimensiones elegidas para el nonograma). También es obvio que la figura de la imagen que se quiere convertir tendrá una conversión más fiel cuanto mejor definida y recalcada esté dicha figura en la imagen (por ejemplo, la digitalización será más fiel para figuras sobre un fondo blanco).

Manual de usuario (interfaz gráfica)

El programa de este proyecto es una aplicación tipo “windows” (ventana) y todo la funcionalidad de este se puede controlar y gestionar mediante una interfaz gráfica y no se hace uso de consola. La aplicación está empaquetada en un fichero Java (.jar), *Hanjie.jar*. Como se indicó en apartados anteriores, la máquina virtual de Java utilizado en el proyecto es **Java 1.6.0_06**. Toda la interfaz gráfica ha sido desarrollada en Netbeans IDE 6.7 con esta versión de Java.

Una vez arrancado el programa, se abrirá el programa “vacío”, como se observa en la *figura 11* . Se ha comentado anteriormente que hay dos formas de cargar nonogramas en la aplicación. La primera de ellas es cargando un fichero de texto plano.

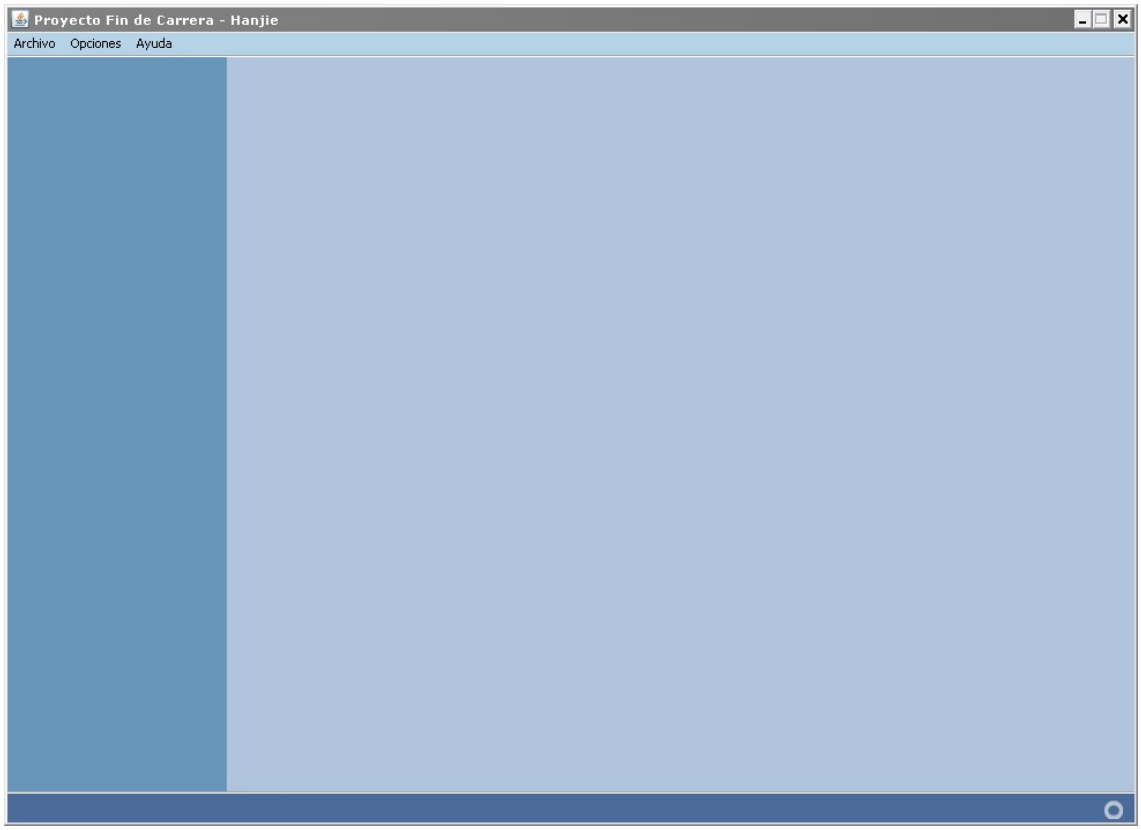


Fig. 11 – Aplicación iniciada.

Para seleccionar un fichero de texto, basta seleccionar el menú “*Archivo*” y pulsar en “*Cargar fichero…*” (*Figura 12*) y se abrirá un explorador de documentos para seleccionar el fichero.

Fig. 12 – Menú Archivo.

El fichero a cargar deberá respetar un formato específico para poder ser aceptado y leído por el programa. Por ejemplo, un fichero de un nonograma de tamaño 5x5 tendría el siguiente formato:

```
#HORIZONTALES
2
1
1,2
2,1
3
#VERTICALES
3
1,2
1,1
1,1
2
```

La primera línea del fichero contendrá la cadena "#HORIZONTALES". En las siguientes líneas del fichero vendrán las entradas horizontales (por filas) del pasatiempo, que serán 5, 10, 15 o 20 líneas. Tras las entradas de filas, la siguiente línea deberá contener la cadena "#VERTICALES" y, a continuación, las entradas verticales, cada una en una fila del fichero.

Cada línea que comprenda una entrada, ya sea horizontal o vertical, tendrá sus bloques separados por una "coma" (,), pudiendo quedar también en blanco, representando una fila o columna sin bloques de celdas negras. Teniendo en cuenta los tamaños de nonogramas soportados por la aplicación, los ficheros de texto tendrán una longitud de 12, 22, 32 o 42 líneas, según se trate de tableros de 5x5, 10x10, 15x15 o 20x20, respectivamente.

Si el fichero no respeta el formato explicado, el programa rechazara la carga del nonograma. En caso de que el fichero sí sea correcto, se inicia la partida para el usuario.

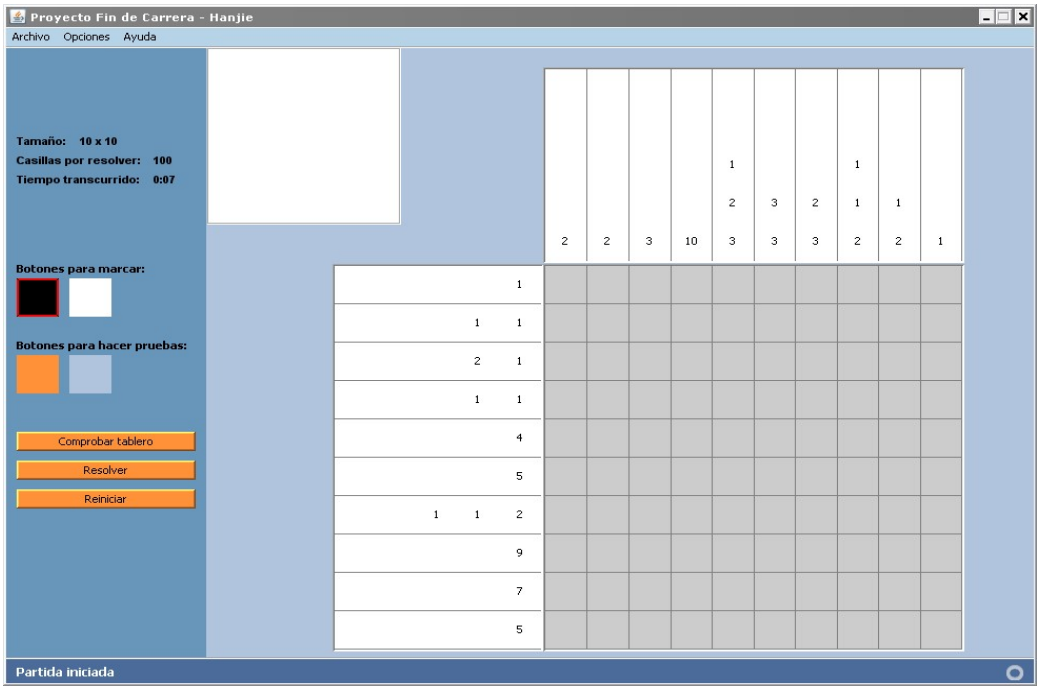


Fig. 13. Partida iniciada.

En el panel principal de la aplicación se encuentra el tablero del juego. También aparece un mini tablero, cuya función es simplemente mostrar la figura que se va formando con el nonograma.

En el panel de la izquierda aparecen información relativa a la partida actual y algunas opciones del juego. En la parte superior, se muestra el tamaño del nonograma en juego, las casillas por resolver que quedan y el tiempo transcurrido desde que se inició la partida.

Un poco más abajo, hay una serie de cuatro botones de colores. Estos botones representan el color a marcar cuando se hace *click* sobre el tablero principal. Por defecto, cuando se inicia la partida, el botón seleccionado es el de color negro, de forma que cuando se hace click en alguna de las celdas del tablero la marcará de negro. Si se quiere marcar celdas en blanco, basta con seleccionar el botón blanco. Los botones naranja y celeste sirven para hacer pruebas sobre el tablero. Esto puede ser útil para tableros complejos, y simular en lugar de marcar las celdas para poder comprobar posibles caminos a la solución (el naranja simularía al negro y el celeste al blanco). Los botones negros y blancos sirven para marcar (o desmarcar) celdas de forma "efectiva".

El botón derecho del ratón se puede usar para marcar, desmarcar o simular celdas de forma que actúa como el botón opuesto al seleccionado en ese momento. Es decir, si está activo el botón negro, el botón derecho del ratón podrá usarse para marcar en blanco; si el activo es el naranja, se podrá usar para simular una celda celeste.

A continuación se dispone de un grupo de 3 botones. El primero de ellos, etiquetado con "*Comprobar tablero*", sirve para comprobar si el estado actual del tablero es correcto o no. El segundo, "*Resolver*", inicia la resolución automática del nonograma cargado. Este botón se explicará con más detalles más adelante. El tercer botón, "*Reiniciar*", inicia la partida de nuevo, limpiando el tablero y reiniciando el contador de tiempo transcurrido.

Fig. 14– Panel izquierdo

Además de las opciones vistas anteriormente, existe más funcionalidad en el menú *Opciones*. Todas las opciones de este menú son *chequeables*, estando algunas activas por defecto al arrancarse el programa.

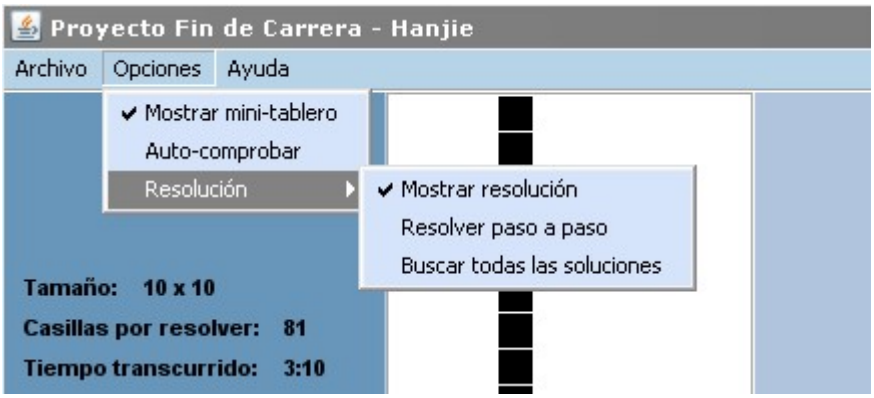


Fig. 15 – Menú Opciones

La primera opción que aparece, y que está activada por defecto, es "Mostrar mini-tablero", muestra o no en el panel principal el mini-tablero que se comentó antes. La segunda opción es "Auto-comprobar". Cuando se activa esta opción, se realiza una comprobación del estado actual del tablero. Mientras esté activa esta opción, esta comprobación se realizará siempre que se marque una celda del tablero (con color blanco o negro, nunca con naranja o celeste). El estado del tablero se muestra junto al resto de información de la partida (*figura 15*).

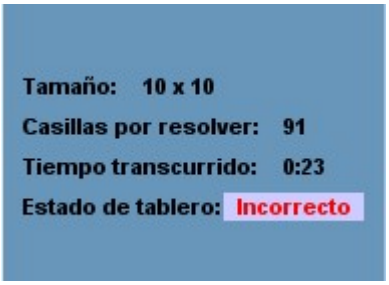


Fig. 16– Auto comprobación

El menú *Opciones* contiene otro grupos de opciones, "*Resolución"*, cuyo contenido es funcionalidad relativa a la resolución automática de los puzzles.

- "*Mostrar resolución"*: cuando esta opción está activa y se pulsa el botón "*Resolver*", se puede ver en el propio tablero principal las transiciones que sufre el puzzle mientras busca la solución. Los cambios que se muestran en el tablero son consecuencia de la ejecución del algoritmo del programa. Con esta opción, el algoritmo tarda más tiempo en resolver el tablero, y en algunos casos la diferencia puede ser considerable. Esto es debido al tiempo que se gasta en redibujar el tablero, que se pinta por completo cuando es modificado.
- "*Resolver paso a paso*": aunque esté marcada, esta opción solo tiene efecto si lo está la anterior ("*Mostrar resolución*"). Activa o desactiva la resolución paso a paso. Cuando pulsamos el botón para resolver el nonograma, se crea un hilo de ejecución paralelo al hilo principal de la aplicación. Con esto es posible poder detener, reanudar, avanzar o incluso cancelar la ejecución de la resolución del tablero. Si se activa esta opción y se pulsa el botón "Resolver", la resolución se detiene tras marcar la primera celda de la solución y aparecen nuevos botones en el panel de la izquierda de la aplicación. (*figura 16*). Si pulsamos "Siguiente paso",

la resolución del puzzle avanza hasta la próxima celda marcada, volviéndose a detener el proceso, y así mientras se vaya pulsando "Siguiente paso" hasta a la solución, o bien, pulsar el botón "Continuar >>", que hace avanzar la resolución hasta llegar a la solución de una vez, sin paradas. También se puede cancelar la búsqueda de solución pulsando el botón "Cancelar resolución", que no sólo está activo cuando se usa la opción de "*Resolución paso a paso*", si no que está disponible siempre que se pulsa el botón "*Resolver*". Como puede verse en la figura de la derecha, los botones "Comprobar tablero", "Resolver" y "Reiniciar" se bloquean durante el proceso de resolución.

Fig. 17 – Resolución paso a paso

- "*Buscar todas las soluciones*": activando esta opción se consiguen todas las soluciones del tablero que se pretende resolver, en el caso de que tuviera más de una. Si finaliza la búsqueda de soluciones y hay más de una solución, es posible visualizar todas las soluciones encontradas mediante unos cursores que aparecerían tras la finalización de la búsqueda, dibujando en el propio tablero las distintas soluciones.

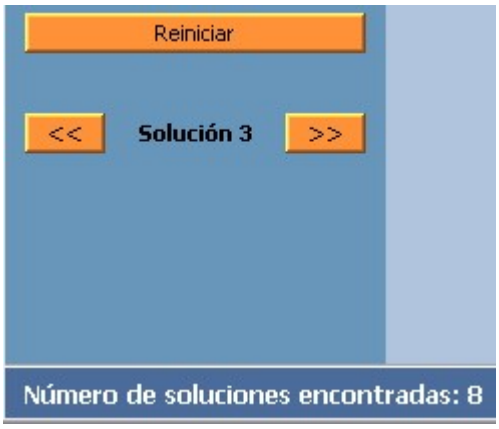


Fig. 18 – Todas las soluciones

En el capitulo anterior se describió el método y proceso de digitalización de imágenes que incorpora la aplicación. Esta es la segunda forma para cargar nonogramas en el programa. Para abrir el convertidor hay que seleccionar "*Convertir imagen…*", dentro del menú "*Archivo*" (*figura 19*).

Fig. 19 – Abriendo el digitalizador de imágenes.

Antes de realizar la conversión para iniciar la partida, es posible previsualizar el nonograma que se generará. Otra posibilidad interesante es la de generar un fichero (botón "*Generar fichero*") de texto con las entradas del nonograma que se generaría, con el formato que se describió líneas atrás en este mismo apartado, de modo que posteriormente se podría cargar con la carga de fichero de texto.

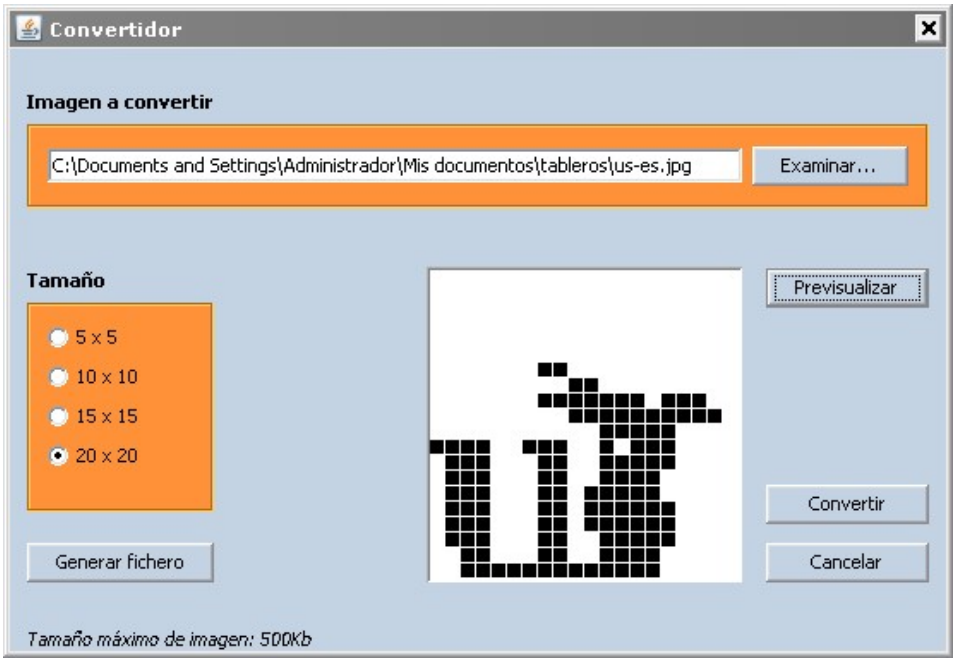


Fig. 20 – Previsualización de una conversión de imagen.

Pulsando "*Convertir*", se inicia la partida de la misma forma que se explicó con la carga de un fichero de texto, pero con una gran salvedad: al generarse la digitalización de la imagen se obtiene también, como es obvio, la solución del tablero. Esta solución se mantiene durante la partida, por lo que la resolución del tablero en este caso no se realiza. En este caso, cuando se pulsa el botón "*Resolver*", lo único que se hace es dibujar la solución que ya se tiene desde el principio. Por lo tanto, las opciones de resolución no tienen efecto, incluso cuando las entradas generadas tienen más de una solución. Si se quiere hacer uso de las opciones del menú "*Resolución*" para aplicarlas al nonograma que se genera con una digitalización, lo apropiado es generar el fichero de texto a partir de la imagen y cargar el fichero generado.

Conclusiones

El tamaño máximo que admite el programa descrito en este proyecto para los nonogramas es 20x20. El algoritmo implementado en la aplicación para buscar soluciones a los puzzles resuelve este problema en un tiempo razonable. Pero si se utilizara este mismo algoritmo para resolver tableros de nonogramas de tamaños mayores, el tiempo de resolución aumentaría considerablemente. Como ya se comentó en anteriores apartados, no existe un algoritmo que solucione todos los nonogramas, sean del tamaño que sean, en un tiempo razonable.

Una de las mayores dificultades encontradas en el desarrollo del proyecto ha sido la incorporación del digitalizador de imágenes. En un principio se optó por adquirir software ya existente y gratuito para integrarlo con la aplicación y adaptarlo a las necesidades que se daban. La complejidad para realizar la integración a las necesidades del proyecto hizo que finalmente se optara por la creación de un convertidor propio del programa.

Esta funcionalidad podría mejorarse con algoritmos más complejos basados en otras medidas estadísticas sobre las imágenes en sí, capaces de reconocer figuras en una imagen de forma más eficiente, pero también consumirían más memoria y tiempo para realizar las transformaciones de esas imágenes a nonogramas.

Un nonograma no deja de ser un pasatiempo, por lo que se ha tratado hacer que el programa sea atractivo para el usuario añadiéndose funcionalidad. Esta funcionalidad podría crecer de cara a ganar en jugabilidad. Por ejemplo, se podrían añadir modos de juego para que el usuario pudiera “medirse” con la máquina, o con otros jugadores (sería interesante mantener un registro de marcas de tiempo por cada nonograma). El hecho de ser un crucigrama basado en imágenes ocultas que hay que descubrir también puede dar mucho juego. Pero estas sugerencias quedan ya lejos del objetivo principal del proyecto.

Fuentes y referencias

Información relativa a los nonogramas:

- http://ma1.eii.us.es/miembros/valvarez/proyectos/hanjie_algoritmo.pdf: algoritmo en el que se basa la aplicación del proyecto.

- http://en.wikipedia.org/wiki/Nonogram: información sobre los puzzles nonogramas en la Wikipedia. Contiene descripciones de las diferentes técnicas para resolverlos.

- http://en.wikipedia.org/wiki/NP-complete: información sobre la complejidad de los nonogramas.

- http://www.puzzlemuseum.com/griddler/gridhist.htm: contiene información sobre los orígenes de los nonogramas.

- http://ende.cc/agujero/rejillas/index.html: web con múltiples ejemplos de tableros de diferentes dimensiones para poder resolver.

- http://www.griddlers.net/pages/home: probablemente la mejor página web sobre nonogramas en Internet.

Información auxiliar:

- http://www.javahispano.org: foros y manuales relativos a Java en castellano.

- http://java.sun.com/docs/books/tutorial/: manuales y tutoriales sobre Java.

- http://es.wikipedia.org/wiki/Modelo_de_color_HSV: en esta entrada de Wikipedia se explica el modelo HSV del color.

- http://help.adobe.com/es_ES/Photoshop/10.0/help.html?content=WSAAFD9CC8-831E-4593-8694-B39919F72A26.html: más información sobre el modelo del color HSV.

Anexo A: Análisis de costes

	Tiempo estimado
Hanjie (parte lógica)	
Análisis	15 horas
Desarrollo	70 horas
Hanjie (interfaz gráfica)	
Desarrollo	80 horas
Digitalizador de imágenes	
Análisis	2 horas
Desarrollo	10 horas
Documentación y Memoria del proyecto	18 horas

Tiempo total estimado en la realización del proyecto = 195 horas

Anexo B: Código fuente

Fuentes del digitalizador de imágenes

ResultadoConvertidor.java

```
package hanjie.convertidor;

public class ResultadoConvertidor {

    private int [][] tablero;
    private int [][] entradasH;
    private int [][] entradasV;
    private int tamaño;

    public ResultadoConvertidor(){}

    public int[][] getTablero() {
        return tablero;
    }

    public void setTablero(int[][] tablero) {
        this.tablero = tablero;
    }

    public int[][] getEntradasH() {
        return entradasH;
    }

    public void setEntradasH(int[][] entradasH) {
        this.entradasH = entradasH;
    }

    public int[][] getEntradasV() {
        return entradasV;
    }

    public void setEntradasV(int[][] entradasV) {
        this.entradasV = entradasV;
    }

    public int getTamaño() {
        return tamaño;
    }

    public void setTamaño(int tamaño) {
        this.tamaño = tamaño;
    }
}
```

Convertidor.java

```
package hanjie.convertidor;

import java.awt.Color;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.Vector;
import javax.imageio.ImageIO;


public class Convertidor
{

   public static boolean convertirImagen(int tamaño, File imagen, ResultadoConvertidor resultado){

      try{
         int [][] tablero = generarTablero(tamaño, imagen);
         int [][] entradasH = new int[tamaño][];
         int [][] entradasV = new int[tamaño][];

         if(generarEntradas(entradasH, entradasV, tablero)){

            resultado.setTablero(tablero);
            resultado.setEntradasH(entradasH);
            resultado.setEntradasV(entradasV);
            resultado.setTamaño(tamaño);
            return true;
         }
         else{
            return false;
         }
      }
      catch(Exception e){
         return false;
      }
   }

   public static int [][] generarTablero(int tamaño, File FileImagen)
   {
      try
      {
         BufferedImage imagen =  ImageIO.read(FileImagen);
         int[][] tablero = new int[imagen.getHeight()][imagen.getWidth()];
         float [][][] hsbTotal = new float [imagen.getHeight()][imagen.getWidth()][3];

         float sumaBrillos = 0;
         float sumaTonos = 0;
         float sumaSaturacion = 0;
         for(int i=0;i<imagen.getHeight();i++)
         {
            for(int j=0;j<imagen.getWidth();j++)
            {
               int pixel = imagen.getRGB(j,i);
               Color colorPixel = new Color(pixel);

               int r = colorPixel.getRed();
               int g = colorPixel.getGreen();
               int b = colorPixel.getBlue();

               float[] hsb = Color.RGBtoHSB(r, g, b, null);
               hsbTotal[i][j] = hsb;
               //hsb[0]: tono
               //hsb[1]: saturacion
```

```
            //hsb[2]: brillo
            sumaTonos += hsb[0];
            sumaSaturacion += hsb[1];
            sumaBrillos += hsb[2];
        }
    }

    float mediaTono = sumaTonos/(imagen.getHeight()* imagen.getWidth());
    float mediaSaturacion = sumaSaturacion/(imagen.getHeight()* imagen.getWidth());
    float mediaBrillos = sumaBrillos/(imagen.getHeight()* imagen.getWidth());

    for(int i=0;i<imagen.getHeight();i++)
    {
        for(int j=0;j<imagen.getWidth();j++)
        {
            float[] hsb = hsbTotal[i][j];

            if(hsb[2] >= mediaBrillos && hsb[1] <= mediaSaturacion){//1.0 es el mayor brillo
                tablero[i][j] = 1;
            }
            else{
                if(hsb[2] >= mediaBrillos){
                    tablero[i][j] = 1;
                }
                else{
                    tablero[i][j] = 2;
                }
            }
        }
    }

    //redimensionamos el tablero al tamaño indicado en el parámetro de entrada
    int bloqueAncho = imagen.getWidth()/tamaño;
    int bloqueAlto = imagen.getHeight()/tamaño;
    int [][] nuevoTablero = new int[tamaño][tamaño];

    //teniendo en cuenta el ancho
    for(int i=0;i<tamaño;i++)
    {
        for(int j=0;j<tamaño;j++)
        {
            int contadorNegros = 0;

            for(int x=0;x<bloqueAlto;x++)
            {
                for(int y=0;y<bloqueAncho;y++)
                {
                    int indX = (i * bloqueAlto) + x;
                    int indY = (j * bloqueAncho) + y;
                    if(tablero[indX][indY] == 2){
                            contadorNegros++;
                    }
                }
            }

            if(contadorNegros>=((bloqueAlto*bloqueAncho)/2))
                nuevoTablero[i][j] = 2;
            else
                nuevoTablero[i][j] = 1;
        }
    }

    return nuevoTablero;

}
catch (IOException ex) {
    ex.printStackTrace();
```

```
            return null;
        }
    }

    private static boolean generarEntradas(int [][] entradasH, int [][] entradasV, int [][] tablero)
    {
        boolean resultado = true;

        //entradasHorizontales
        for(int i=0;i<tablero.length;i++){
            Vector vectorHorizontales = new Vector();
            for(int j=0;j<tablero.length;j++){
                if(tablero[i][j] == 2){ //encuentra un nuevo bloque
                    int x = j;
                    int tamañoBloque = 0;
                    while(x < tablero.length && tablero[i][x] == 2){
                        tamañoBloque++;
                        x++;
                    }
                    vectorHorizontales.add(tamañoBloque);
                    j = x;
                }
            }
            if(vectorHorizontales.size() == 0)
                entradasH[i] = null;
            else
                entradasH[i] = new int[vectorHorizontales.size()];
            for(int ind=0;ind<vectorHorizontales.size();ind++){
                    entradasH[i][ind] = (Integer) vectorHorizontales.get(ind);
            }
        }
        //entradasVerticales
        for(int i=0;i<tablero.length;i++){
            Vector vectorVerticales = new Vector();
            for(int j=0;j<tablero.length;j++){
                if(tablero[j][i] == 2){ //encuentra un nuevo bloque
                    int x = j;
                    int tamañoBloque = 0;
                    while(x < tablero.length && tablero[x][i] == 2){
                        tamañoBloque++;
                        x++;
                    }
                    vectorVerticales.add(tamañoBloque);
                    j = x;
                }
            }
            if(vectorVerticales.size()==0)
                entradasV[i] = null;
            else
                entradasV[i] = new int[vectorVerticales.size()];
            for(int ind=0;ind<vectorVerticales.size();ind++){
                    entradasV[i][ind] = (Integer) vectorVerticales.get(ind);
            }
        }
        return resultado;
    }
}
```

Fuentes de la resolución de nonogramas

Hanjie.java

```
package hanjie.alg.principal;

import hanjie.HanjieAplicacionView;
import hanjie.alg.elementos.ArrayCeldas;
import hanjie.alg.util.Estados;
import hanjie.alg.elementos.Celda;
import hanjie.alg.elementos.TableroHanjie;
import java.awt.Color;
import java.util.Vector;
import java.util.logging.Level;
import java.util.logging.Logger;



public class Hanjie {

   public static boolean simula = false;


   public static TableroHanjie solucionRecursiva(TableroHanjie tablero, int [][] entradasH, int [][]
entradasV, int profundidad, HanjieAplicacionView hav)
   {
      TableroHanjie tableroAux = null;
      int tamañoJuego = tablero.getTamaño();

      if(esSolucion(tablero)){
         hav.soluciones.add(tablero);
         return tablero;
      }
      else{
         boolean cambioDeColor = false;

         tableroAux = new TableroHanjie(tablero);
         buscaYMarca(tableroAux, Estados.BLANCA, hav);
         simpleBoxesSpaces(tableroAux, tamañoJuego, entradasH, entradasV, hav);
         compruebaTablero(tableroAux, entradasH, entradasV);
         if(tableroAux.isCorrecto()){
            tableroAux = new TableroHanjie(solucionRecursiva(tableroAux, entradasH, entradasV,
profundidad + 1, hav));
            compruebaTablero(tableroAux, entradasH, entradasV);
            if(hav.buscarTodasSoluciones == true){
               cambioDeColor = true;
            }
            else{
               if(!tableroAux.isCorrecto())
                  cambioDeColor = true;
            }
         }
         else{
            cambioDeColor = true;
         }

         if(cambioDeColor){
            tableroAux = new TableroHanjie(tablero);  //deshago el camino recorrido marcando "la casilla"
sin estado
            buscaYMarca(tableroAux, Estados.NEGRA, hav);    //y la marca en negro
            simpleBoxesSpaces(tableroAux, tamañoJuego, entradasH, entradasV, hav);
```

```
            compruebaTablero(tableroAux, entradasH, entradasV);

            if(tableroAux.isCorrecto()){
               tableroAux = new TableroHanjie(solucionRecursiva(tableroAux, entradasH, entradasV,
profundidad + 1, hav));
            }
         }
      }

      return tableroAux;
   }



   public static boolean buscaYMarca(TableroHanjie tablero, int tipo, HanjieAplicacionView hav)
   {
      int iFinal = 0;
      int jFinal = 0;

      int tamaño = tablero.getTamaño();
      boolean encontrado = false;

         //buscamos por filas (lo podríamos hacer por columnas...)
      for(int i=0;i<tamaño && !encontrado;i++){
         for(int j=0;j<tamaño && !encontrado;j++){
            Celda celda = tablero.getFilas()[i].getCeldas()[j];
            if(celda.getEstado() == Estados.SIN_COLOR)
            {
               iFinal = i;
               jFinal = j;
               encontrado = true;
            }
         }
      }
      if(encontrado){
         return marcarCelda(iFinal, jFinal, tablero, tipo, hav);
      }
      else{
         return false;
      }
   }


   //Implementación de Simple Boxes y Simple Spaces.
   public static boolean simpleBoxesSpaces(TableroHanjie tablero, int tamañoJuego, int [][] entradasH,
int [][] entradasV, HanjieAplicacionView hav)
   {
      boolean hayCambios = true;
      int iteracion = 0;
      while(hayCambios)
      {
         compruebaTablero(tablero, entradasH, entradasV);
         if(!tablero.isCorrecto())
               return false;

         hayCambios = false;
         iteracion++;
         //HORIZONTALES           /
         for(int i=0; i<entradasH.length;i++)
         {
            int entradaH [] = entradasH[i];
            hayCambios = simpleBoxesSpacesPorFila(entradaH, tamañoJuego, tablero, hayCambios, i,
hav) || hayCambios;
         }

         //VERTICALES
         for(int j=0; j<entradasV.length;j++)
```

```
            {
                int entradaV [] = entradasV[j];
                hayCambios = simpleBoxesSpacesPorColumna(entradaV, tamañoJuego, tablero, hayCambios,
j, hav) || hayCambios;
            }
        }


        return hayCambios;
    }

    public static boolean simpleBoxesSpacesPorFila(int [] entradaH, int tamañoJuego, TableroHanjie
tablero, boolean hayCambios, int i, HanjieAplicacionView hav)
    {
        //calculamos las combinaciones
        Vector filaAux = new Vector();
        for(int x = 0;x<tamañoJuego; x++){
            filaAux.add(x,0);
        }
        Vector combinacionesFila = new Vector();
        obtenerCombinaciones(0, entradaH, filaAux, combinacionesFila);

        //de las combinaciones obtenidas, seleccionamos aquellas que
        //son factibles en función del estado actual del tablero
        Vector aux = new Vector (combinacionesFila);
        combinacionesFila.clear();
        for(int j=0; j < aux.size(); j++){
            Vector fila = (Vector) aux.elementAt(j);
            if(aceptaFila(i, entradaH, tablero, fila)){
                combinacionesFila.add(fila);
            }
        }

        boolean [] filaBlancas = new boolean[tamañoJuego];
        boolean [] filaNegras = new boolean[tamañoJuego];
        for(int x=0; x < tamañoJuego; x++){
            filaBlancas[x] = true;
            filaNegras[x] = true;
        }

        int numFilasPosibles = combinacionesFila.size();

        for(int j=0; j < numFilasPosibles; j++){
            Vector fila = (Vector) combinacionesFila.elementAt(j);
            for(int x=0; x<tamañoJuego; x++){
                Object celda = fila.get(x);
                if(celda.equals(1)){
                    filaBlancas[x] = false;
                }
                else if(celda.equals(0)){
                    filaNegras[x] = false;
                }
            }
        }

        for(int j=0; j<tamañoJuego; j++){
            if(filaNegras[j] == true ){
                hayCambios = marcarCelda(i, j, tablero, Estados.NEGRA, hav) || hayCambios;
            }
            if(filaBlancas[j] == true ){
                hayCambios = marcarCelda(i, j, tablero, Estados.BLANCA, hav) || hayCambios;
            }
        }

        return hayCambios;
    }
```

```
  public static boolean simpleBoxesSpacesPorColumna(int [] entradaV, int tamañoJuego,
TableroHanjie tablero, boolean hayCambios, int j, HanjieAplicacionView hav)
  {
     //calculamos las combinaciones
     Vector columnaAux = new Vector();
     for(int x = 0;x<tamañoJuego; x++){
          columnaAux.add(x,0);
     }
     Vector combinacionesColumna = new Vector();
     obtenerCombinaciones(0, entradaV, columnaAux, combinacionesColumna);

     //de las combinaciones obtenidas, seleccionamos aquellas
     //que son factibles en función del estado actual del tablero
     Vector aux = new Vector (combinacionesColumna);
     combinacionesColumna.clear();
     for(int i=0; i < aux.size(); i++){
        Vector columna = (Vector) aux.elementAt(i);
        if(aceptaColumna(j, entradaV, tablero, columna)){
             combinacionesColumna.add(columna);
        }
     }

     boolean [] columnaBlancas = new boolean[tamañoJuego];
     boolean [] columnaNegras = new boolean[tamañoJuego];
     for(int x=0; x < tamañoJuego; x++){
        columnaBlancas[x] = true;
        columnaNegras[x] = true;
     }

     int numColumnasPosibles = combinacionesColumna.size();

     for(int i=0; i < numColumnasPosibles; i++){
        Vector columna = (Vector) combinacionesColumna.elementAt(i);
        for(int x=0; x<tamañoJuego; x++){
           Object celda = columna.get(x);
           if(celda.equals(1)){
              columnaBlancas[x] = false;
           }
           else if(celda.equals(0)){
              columnaNegras[x] = false;
           }
        }
     }

     for(int i=0; i<tamañoJuego; i++){
        if(columnaNegras[i] == true ){
           hayCambios = marcarCelda(i, j, tablero, Estados.NEGRA, hav) || hayCambios;
        }
        if(columnaBlancas[i] == true ){
           hayCambios = marcarCelda(i, j, tablero, Estados.BLANCA, hav) || hayCambios;
        }
     }

     return hayCambios;
  }



  public static void obtenerCombinaciones(int indEntrada,  int [] entrada, Vector vector, Vector
combinaciones)
  {
     if(entrada == null || entrada.length <=0){  //FILA o COLUMNA TOTALMENTE EN BLANCO
        Vector vectorAux = new Vector(vector);
        for(int i=0;i<vector.size();i++)
           vectorAux.set(i,0);
        combinaciones.add(vectorAux);
        return;
```

```
        }
        int n = entrada[indEntrada];
        int posicion = vector.lastIndexOf(1);

        if(posicion == -1)
            posicion = 0;
        else
            posicion = posicion + 2;

        for(int x=0;x + posicion + n<=vector.size();x++){

            Vector vectorAux = new Vector(vector);
            for(int i=0;i<n;i++)
                vectorAux.set(posicion + x + i,1);

            if(indEntrada==entrada.length-1)
                combinaciones.add(vectorAux);
            else
                obtenerCombinaciones(indEntrada+1, entrada, vectorAux, combinaciones);
        }
    }


    public static boolean aceptaFila(int i, int [] entradaH, TableroHanjie tablero, Vector fila)
    {
        boolean aceptado=true;
        Celda [] f = tablero.getFilas()[i].getCeldas();
        for(int x=0; x<f.length && aceptado;x++){
            int estado = (Integer) fila.get(x);
            int estadoCelda = f[x].getEstado();
            //el vector fila contiene 1's (celda negra) y 0's (celda blanca)
            if((estadoCelda == Estados.BLANCA && estado == 1) || (estadoCelda == Estados.NEGRA &&
estado == 0)){
                aceptado = false;
            }
        }
        return aceptado;
    }


    public static boolean aceptaColumna(int j, int [] entradaV, TableroHanjie tablero, Vector columna)
    {
        boolean aceptado=true;
        Celda [] f = tablero.getColumnas()[j].getCeldas();
        for(int x=0; x<f.length && aceptado;x++){
            int estado = (Integer) columna.get(x);
            int estadoCelda = f[x].getEstado();
            if((estadoCelda == Estados.BLANCA && estado == 1) || (estadoCelda == Estados.NEGRA &&
estado == 0)){
                    aceptado = false;
            }
        }
        return aceptado;
    }

    public static boolean marcarCelda(int fila, int columna, TableroHanjie tablero, int tipo,
HanjieAplicacionView hav)
    {
        boolean hayCambios = false;
        Celda celda = new Celda();
        celda.setEstado(tipo);
        if(tablero.getFilas()[fila].getCeldas()[columna].getEstado() != tipo) //tb se podria comprobar con las
Columnas
        {
            if(tipo == Estados.SIN_COLOR){
                tablero.setCeldasSinResolver(tablero.getCeldasSinResolver()+1);
            }
```

```
            else{
                if(tablero.getFilas()[fila].getCeldas()[columna].getEstado() == Estados.SIN_COLOR)
                    tablero.setCeldasSinResolver(tablero.getCeldasSinResolver()-1);
            }

            tablero.getFilas()[fila].getCeldas()[columna] = new Celda(celda);
            tablero.getColumnas()[columna].getCeldas()[fila] = new Celda(celda);

            hayCambios = true;
        }

        //DIBUJO TABLA
        if(hayCambios && hav!=null && hav.jCheckBoxMenuItemMostrarResolución.isSelected()){
            for(int i=0;i<tablero.getTamaño();i++){
                for(int j=0;j<tablero.getTamaño();j++){
                    if(tablero.getFilas()[i].getCeldas()[j].getEstado() == Estados.BLANCA){
                        if(!simula){
                            hav.jTablero.getModel().setValueAt(Color.WHITE, i, j);
                            hav.jTableroMini.getModel().setValueAt(Color.WHITE, i, j);
                        }
                        else{
                            hav.jTablero.getModel().setValueAt(hav.BlancoFalso, i, j);
                            hav.jTableroMini.getModel().setValueAt(hav.BlancoFalso, i, j);
                        }
                    }
                    else if(tablero.getFilas()[i].getCeldas()[j].getEstado() == Estados.NEGRA){
                        if(!simula){
                            hav.jTablero.getModel().setValueAt(Color.BLACK, i, j);
                            hav.jTableroMini.getModel().setValueAt(Color.BLACK, i, j);
                        }
                        else{
                            hav.jTablero.getModel().setValueAt(hav.NegroFalso, i, j);
                            hav.jTableroMini.getModel().setValueAt(hav.NegroFalso, i, j);
                        }
                    }
                    else if(tablero.getFilas()[i].getCeldas()[j].getEstado() == Estados.SIN_COLOR){
                        hav.jTablero.getModel().setValueAt(new Color(204,204,204), i, j);
                        hav.jTableroMini.getModel().setValueAt(Color.WHITE, i, j);
                    }
                }
            }
            if(hav.jCheckBoxMenuItemPasoAPaso.isSelected() && !hav.continuarResolver){
                try {
                    hav.hrt.suspend();
                } catch (Exception ex) {
                    Logger.getLogger(Hanjie.class.getName()).log(Level.SEVERE, null, ex);
                }
            }
        }

        return hayCambios;
    }

    //Comprueba si el tablero contiene alguna contradicción
    public static void compruebaTablero(TableroHanjie tablero, int [][] entradasH, int [][] entradasV)
    {
        tablero.setCorrecto(true);
        int tamaño = tablero.getTamaño();
        //FILAS:
        //primero comprobamos si las filas son correctas
        for(int i=0;i<tamaño;i++)
        {
            int [] entradaH = entradasH[i];
            int sumaFilaEntrada = 0;
            if(entradaH == null) // Fila en blanco
                entradaH = new int[0];
```

```
        for(int x=0;x<entradaH.length;x++)
           sumaFilaEntrada += entradaH[x];

        int sumaFilaTablero = 0;
        ArrayCeldas filaCeldas = tablero.getFilas()[i];

        for(int j=0;j<tamaño;j++){
           if(filaCeldas.getCeldas()[j].getEstado() == Estados.NEGRA)
              sumaFilaTablero++;
        }

        //Se pueden dar 3 casos:
        // - Que sumaFilaTablero sea mayor que sumaFilaEntrada, siendo el tablero incorrecto
        // - Que sean iguales, pudiendo ser correcto o no, dependiendo de la distribución de las celdas
Negras
        //    (aún siendo coherente esta distribución, PODRÍA ser un tablero incorrecto). Podría tratarse
también
        //   de una fila "completa"(Ver parte final de la función CompruebaDistribucionFila) (correcta o
        //   incorrecta, eso se vería más adelante)
        // - Que sumaFilaTablero sea menor que sumaFilaEntrada, ocurre lo mismo que en el caso
anterior
        //   (salvo de que se trate de una fila "completa")

        if(sumaFilaTablero > sumaFilaEntrada){
           tablero.setCorrecto(false);
           return;
        }
        else if(sumaFilaTablero == sumaFilaEntrada || sumaFilaTablero < sumaFilaEntrada){
           if(!compruebaDistribucionFila(tablero, entradaH, i)){
              tablero.setCorrecto(false);
              return;
           }
        }
     }

     //COLUMNAS:
     //lo mismo que antes hicimos con las filas
     for(int j=0;j<tamaño;j++)
     {
        int [] entradaV = entradasV[j];
        int sumaColumnaEntrada = 0;

        if(entradaV == null) // Columna en blanco
           entradaV = new int[0];


        for(int x=0;x<entradaV.length;x++)
           sumaColumnaEntrada += entradaV[x];

        int sumaColumnaTablero = 0;
        ArrayCeldas columnaCeldas = tablero.getColumnas()[j];

        for(int x=0;x<tamaño;x++){
           if(columnaCeldas.getCeldas()[x].getEstado() == Estados.NEGRA)
              sumaColumnaTablero++;
        }

        //Se pueden dar 3 casos:
        // - Que sumaColumnaTablero sea mayor que sumaColumnaEntrada, siendo el tablero incorrecto
        // - Que sean iguales, pudiendo ser correcto o no, dependiendo de la distribución de las celdas
Negras
        //    (aún siendo coherente esta distribución, PODRÍA ser un tablero incorrecto). Podría tratarse
también
        //    de una columna "completa" (Ver parte final de la función CompruebaDistribucionColumna)
(correcta o
        //   incorrecta, eso se vería más adelante)
```

```
        // - Que sumaFilaTablero sea menor que sumaFilaEntrada, ocurre lo mismo que en el caso anterior
        //   (salvo de que se trate de una fila "completa")

        if(sumaColumnaTablero > sumaColumnaEntrada){
           tablero.setCorrecto(false);
           return;
        }
        else if(sumaColumnaTablero == sumaColumnaTablero || sumaColumnaTablero < sumaColumnaTablero){
           if(!compruebaDistribucionColumna(tablero, entradaV, j)){
              tablero.setCorrecto(false);
              return;
           }
        }
     }
  }


  private static boolean compruebaDistribucionFila(TableroHanjie tablero, int [] entradaH, int i)
  {
     //calculamos las combinaciones
     Vector filaAux = new Vector();
     for(int x = 0;x<tablero.getTamaño(); x++){
        filaAux.add(x,0);
     }
     Vector combinacionesFila = new Vector();
     obtenerCombinaciones(0, entradaH, filaAux, combinacionesFila);


     //de las combinaciones obtenidas, seleccionamos aquellas que son factibles en función del
     //estado actual del tablero
     Vector aux = new Vector (combinacionesFila);
     combinacionesFila.clear();
     for(int x=0; x < aux.size(); x++){
        Vector fila = (Vector) aux.elementAt(x);
        if(aceptaFila(i, entradaH, tablero, fila)){
           combinacionesFila.add(fila);
        }
     }

     //Si no se acepta ninguna de las combinaciones posibles, es que hemos encontrado una contradicción
     if(combinacionesFila.size() == 0){
        return false;
     }
     //en otro caso, no existe contradicción y es posible que la fila comprobada sea correcta
     else{
        //en el caso de que solo haya una fila posible, comprobamos si es completa para marcarla como tal
        if(combinacionesFila.size() == 1){
           boolean esFilaCompleta = true;
           for(int x=0;x<tablero.getTamaño() && esFilaCompleta;x++){
              if(tablero.getFilas()[i].getCeldas()[x].getEstado() == Estados.SIN_COLOR)
                 esFilaCompleta = false;
           }
           tablero.getFilas()[i].setCompletado(esFilaCompleta);
        }

        return true;
     }
  }

  private static boolean compruebaDistribucionColumna(TableroHanjie tablero, int [] entradaV, int j)
  {
     //calculamos las combinaciones
     Vector columnaAux = new Vector();
```

```
        for(int x = 0;x<tablero.getTamaño(); x++){
            columnaAux.add(x,0);
        }
        Vector combinacionesColumna = new Vector();
        obtenerCombinaciones(0, entradaV, columnaAux, combinacionesColumna);


        //de las combinaciones obtenidas, seleccionamos aquellas
        //que son factibles en función del estado actual del tablero
        Vector aux = new Vector (combinacionesColumna);
        combinacionesColumna.clear();
        for(int x=0; x < aux.size(); x++){
            Vector columna = (Vector) aux.elementAt(x);
            if(aceptaColumna(j, entradaV, tablero, columna)){
                combinacionesColumna.add(columna);
            }
        }

        //Si no se acepta ninguna de las combinaciones posibles, es que hemos encontrado una
contradicción
        if(combinacionesColumna.size() == 0){
            return false;
        }
        //en otro caso, no existe contradicción y es posible que la fila comprobada sea correcta
        else{
            //en el caso de que solo haya una fila posible, comprobamos si es completa para marcarla como
tal
            if(combinacionesColumna.size() == 1){
                boolean esColumnaCompleta = true;
                for(int x=0;x<tablero.getTamaño() && esColumnaCompleta;x++){
                    if(tablero.getColumnas()[j].getCeldas()[x].getEstado() == Estados.SIN_COLOR)
                        esColumnaCompleta = false;
                }
                tablero.getColumnas()[j].setCompletado(esColumnaCompleta);
            }

            return true;
        }
    }

    public static boolean esSolucion(TableroHanjie tablero)
    {
        return tablero.getCeldasSinResolver() == 0 && tablero.isCorrecto();
    }
}
```