Memoria práctica opcional

Sistemas inteligentes curso
2019-2020

ALUMNO/A: Alejandro Ruiz Valle
DNI: 26804359K
GRUPO: 2ºA Ingeniería Informática
E-MAIL: alexviola@uma.es

1. Resumen y descripción del proyecto

La idea principal de este proyecto consiste en realizar una pequeña IA capaz de resolver un problema concreto que queramos resolver. En nuestro caso, hemos decidido basar nuestro proyecto en los famosos sudokus.

El sudoku es un juego matemático basado en la colocación de números que se popularizó en Japón en 1986. Para ello partimos de una cuadrícula a rellenar de 9x9, dividida en cuadrículas más pequeñas de 3x3. Debemos colocar las cifras del 1 al 9 partiendo de algunos números ya colocados. Como norma, no se puede repetir ninguna cifra en la misma fila, columna o cuadrícula de 3x3.

Para realizar este proyecto, hemos utilizado el lenguaje Java en su versión JavaSE-1.7 y el entorno de desarrollo Eclipse IDE.

2. Objetivos

Con nuestro programa queremos conseguir que la pequeña IA sea capaz, mediante un algoritmo de backtracking, de resolver cualquier cuadrícula 9x9 que pasemos como parámetro.

El algoritmo de Backtracking o vuelta atrás se usa para hacer búsquedas a través de todas las configuraciones posibles en un espacio con restricciones. En nuestro caso, usaremos el algoritmo para encontrar una posible solución dadas las restricciones iniciales del problema.

3. Metodología y explicación de código

La metodología utilizada en este proyecto se basa en tres fases: diseño estructural de la aplicación, desarrollo software, pruebas o testing.

Diseño estructural

En esta fase el objetivo principal era dividir o crear las clases principales que hacen funcionar a nuestro programa. Dando una estructura final formada por 4 clases. Estas clases debían resolver las siguientes necesidades específicas:

- Tablero: definiría todos los parámetros relacionados con el tablero de juego, es decir el tamaño, cantidad de celdas, comprobaciones, impresión del tablero,etc.

- IA: Esta clase implementaría el algoritmo encargado de la resolución del tablero con todos los métodos necesarios.

- Caja: En las prácticas y problemas de clase, vimos siempre resoluciones de búsqueda hechas con nodos. En nuestro sudoku no tenemos nodos como tal, por tanto, decidimos utilizar cada celda o caja como uno. Dando al algoritmo la posibilidad de comprobar y moverse en todas las direcciones o donde hubiese otro nodo.

- Main: en nuestra clase main introduciremos el tablero a resolver, recogeremos los tiempos e interactuamos con el resto de clases para resolver el problema.

Desarrollo Software

A continuación vamos a explicar en detalle el funcionamiento de cada clase con profundidad. Comenzamos por la clase tablero.

```java
public class Tablero {


    // El tablero del sudoku esta dividido en celdas o cajas de 9x9
    // por lo que necesitaremos crear la variable de la caja
    //Vamos a necesitar tambien la clase CAJA para definirla

    Caja tabla[][] = new Caja[9][9];

    public Tablero(ArrayList<Integer> SudokuMapa) {

        //Comprobamos que sea 9x9 el tablero

        if (SudokuMapa.size() == 81 ) {

            int contador=0; // Necesitamos aux para dividir el mapa en cajas

            for (int i=0; i < 9; ++i) {

                for(int j=0; j <9; ++j) {

                    int aux = SudokuMapa.get(contador);
                    tabla[i][j] = new Caja();
                    tabla[i][j].setValor(SudokuMapa.get(contador));
                    tabla[i][j].setCoordenadas(i,j);
                    contador++;

                }
            }


        } else {

            System.out.println("Sudoku no valido, recuerda que debe ser 9x9");

        }

        printTablero();
    }


    public void printTablero() {

        for(int i=0; i <9; ++i) {

            if ( i==3 || i == 6 || i == 9 ) {  //Si la altura  es igual a i
                System.out.println("                                        ");

                // Pintamos en blanco para separar cada 3 lineas
            }




            for (int j=0; j <9; j++) {

                if( j==2 || j==5 || j==8) {

                    System.out.print(tabla[i][j].getValor()+ " || "); // Con esto "dibujamos" las cajas, cada 3 hab

                } else {

                    System.out.print(tabla[i][j].getValor()+ " | ");

                }
            }

            System.out.println();
        }
    }


    public Caja[][] Pasartabla(){

        return tabla;
    }

    public boolean resuelto() {

        boolean resuelto = true;

        for ( int i= 0; i < 9; i++) {

            for (int j=0; j < 9; j++) {

                if (tabla[i][j].getValor()==0) {

                    resuelto = false;

                }
            }
```

En nuestra clase tablero lo más importante es comprobar que el sudoku que pasamos como parámetro sea un 9x9 que contenga 81 posiciones. A su vez, dividirá las celdas o cajas dándoles sus coordenadas de posición, interpretadas como i y j.

También tenemos el método printTablero, que nos dibujará en consola una representación gráfica del tablero pasado como parámetro.

Una vez definido el tablero, debemos definir la clase Caja, que será aquella que funcione a modo de “nodos”.

```
public class Caja {


    // Las cajas o celdas estan formadas por diferentes variables
    //  columna, fila y el valor que contienen

private int fila, columna, valor;

private ArrayList<Integer> NoValidos= new ArrayList<Integer>();

public Caja(int fila, int columna, int valor) {

    this.fila= fila;
    this.columna= columna;
    this.valor = valor;

}

public Caja() { } // Constructor sin parametros



// Metodos necesarios que hemos usado en tablero y algunos
public void setValor(int valor) {
                                    // Colocamos el valor de la variable valor
    this.valor = valor;

}

public int getValor() {     // Nos devuelve el valor  de la variable

        return valor;

}


public int getColumna() {

    return columna;                 // Nos devuelve la fila y la columna
}

public int getFila() {

    return fila;


}
```

```java
public void setCoordenadas(int fila, int columna) {
                                                    // Metodo para establecer coordenadas de posicion
    this.fila = fila;
    this.columna = columna;
}


public void setInvalidos(ArrayList<Integer> NoValidos) {

    this.NoValidos=NoValidos;

}
public ArrayList<Integer> getInvalidos(){

    return NoValidos;
}
```

Nuestra Caja es un objeto formado por el valor de las filas, el valor de la columna y el valor que contiene la caja. Este valor tiene que estar entre 1-9 sí aparece como dato en el tablero y si es una caja vacía aparecerá un 0.

A su vez, creamos los diferentes métodos que operarán sobre la clase Caja. Entre ellos tenemos setValor (añadirá un valor a la caja), getValor (retornará el valor de la caja), getColumna (retornará el valor de la columna), getFila (devuelve el valor de la fila), setCoordenadas (colocará la fila y la columna como coordenadas de la caja), setInvalidos (mediante un ArrayList iremos guardando los valores no válidos para esa caja, ya sea porque aparecen colocados en el sudoku inicial o porque más adelante la IA nos indique que es un valor inválido.

Una vez definida la clase Caja y tablero, pasamos a definir la clase IA o nuestro algoritmo encargado de resolver el problema.

```java
public class IA {

    Caja[][] tabla = new Caja[9][9];

    public IA(Caja tabla[][]) {

        this.tabla = tabla;
    }


    // Empezaremos buscando los movimientos no validos o cajas ya ocupadas


    public void EncontrarInvalidos() {


        for (int x = 0; x < 9; ++x) {  // por cada fila y columna hacemos

            for (int y = 0; y < 9; ++y ) {

                ArrayList<Integer> NoValidos = new ArrayList<Integer>();

                Caja c = tabla[x][y];
                final int fila = x;
                final int columna = y;

                for (int k = 0; k < 9; k++ ) { // para todos los valores de la misma fila

                    int valor = tabla[fila][k].getValor();
                    if (valor != 0 && !NoValidos.contains(valor)) {

                        NoValidos.add(valor);
                    }   // Asi añadimos al array de no validos los valores de las filas
                }

                for (int m=0; m<9; m++) { // esto seria para las columnas

                    int valor = tabla[m][columna].getValor();
                    if (valor != 0 && !NoValidos.contains(valor)) {

                        NoValidos.add(valor);

                    }
                }

                CheckRejilla(fila, columna, NoValidos); // tenemos que crear el metodo para comprobar las siguientes
                c.setInvalidos(NoValidos);
```

En esta primera captura vemos como la idea inicial del algoritmo es ir encontrando los valores que hemos pasado como parámetro e ir guardándolos en NoValidos. Estos valores no pueden ser válidos para toda la fila o la columna, es decir, una vez aparezca un número no puede repetirse ni dentro de la cuadrícula 3x3, dentro de la misma fila o dentro de la misma columna.

Para finalizar, invocamos CheckRejilla. Este método lo que hará será comprobar la rejilla 3x3, para lo que tenemos que crear una serie de bucles if para cada posición dentro de la cuadrícula.

```java
    public void CheckRejilla(int fila, int columna, ArrayList<Integer> NoValidos) {

        // Tenemos que crear 1 if por cada posible posicion en la rejilla de 3x3

        if ( fila <= 2 && columna <= 2 ) { // Esquina arriba izquierda

            for ( int i = 0; i <= 2; i++) {
                for ( int j=0; j <=2; ++j) {

                    Caja c = tabla[i][j];
                    if(!NoValidos.contains(c.getValor()) && c.getValor() != 0) {

                        NoValidos.add(c.getValor());
                    }
                }
            }

        }


if ( fila > 5 && columna <= 2 ) { // Esquina abajo izquierda

            for ( int i = 6; i <= 8; i++) {
                for ( int j=0; j <=2; ++j) {

                    Caja c = tabla[i][j];
                    if(!NoValidos.contains(c.getValor()) && c.getValor() != 0) {

                        NoValidos.add(c.getValor());
                    }
                }
            }

        }

if ( fila <= 5 && fila > 2 && columna <= 2 ) { // Centro izquierda

    for ( int i = 3; i <= 5; i++) {
        for ( int j=0; j <=2; ++j) {

            Caja c = tabla[i][j];
            if(!NoValidos.contains(c.getValor()) && c.getValor() != 0) {

                NoValidos.add(c.getValor());
            }
        }
    }

}
```

```
if ( fila <= 2 && columna > 5 ) { // Esquina arriba derecha

    for ( int i = 0; i <= 2; i++) {
        for ( int j=6; j <=8; ++j) {

            Caja c = tabla[i][j];
            if(!NoValidos.contains(c.getValor()) && c.getValor() != 0) {

                NoValidos.add(c.getValor());
            }
        }
    }

}

if ( fila > 5 && columna > 5 ) { // Esquina abajo derecha

    for ( int i = 6; i <= 8; i++) {
        for ( int j=6; j <=8; ++j) {

            Caja c = tabla[i][j];
            if(!NoValidos.contains(c.getValor()) && c.getValor() != 0) {

                NoValidos.add(c.getValor());
            }
        }
    }

}

if ( fila  >= 3 && fila <=5 && columna > 5 ) { // centro derecha

    for ( int i = 3; i <= 5; i++) {
        for ( int j=6; j <=8; ++j) {

            Caja c = tabla[i][j];
            if(!NoValidos.contains(c.getValor()) && c.getValor() != 0) {

                NoValidos.add(c.getValor());
            }
        }
    }

}

if ( fila <=2 && columna > 2 && columna <= 5  ) { // centro arriba

    for ( int i = 0; i <= 2; i++) {
        for ( int j=3; j <=5; ++j) {

            Caja c = tabla[i][j];
            if(!NoValidos.contains(c.getValor()) && c.getValor() != 0) {
```

```
                NoValidos.add(c.getValor());
            }
        }
    }

}

if ( fila > 5 && columna > 2 && columna <= 5 ) { // centro abajo

    for ( int i = 6; i <= 8; i++) {
        for ( int j=3; j <=5; ++j) {

            Caja c = tabla[i][j];
            if(!NoValidos.contains(c.getValor()) && c.getValor() != 0) {

                NoValidos.add(c.getValor());
            }
        }
    }

}

if ( fila >2 && fila <= 5 && columna > 2 && columna <=5 ) { // centro centro

    for ( int i = 3; i <= 5; i++) {
        for ( int j=3; j <=5; ++j) {

            Caja c = tabla[i][j];
            if(!NoValidos.contains(c.getValor()) && c.getValor() != 0) {

                NoValidos.add(c.getValor());
            }
        }
    }

}


    }

// Ahora necesitamos el metodo IA.FUNCIONA que aparece en main

public boolean funciona() {

    boolean hecho = true;

        // vamos a necesitar unas variables que almacene las restricciones

         int maxRest = 0;
         int Restfila, RestColumna;

         Caja tablaRest = new Caja();

         for ( int a = 0; a <= 8; a++) {

             for ( int b = 0; b <=8 ; ++b) {

                 Caja caja=tabla[a][b];
                 if(caja.getValor()==0) {

                     hecho = false;

                     if(caja.getInvalidos().size() > maxRest )
                         maxRest=caja.getInvalidos().size();

                     tablaRest = caja;
                 }
             }
         }


         if (hecho)

             return true;


             ArrayList<Integer> NoValidos = tablaRest.getInvalidos();

             if (NoValidos.contains(1) && NoValidos.contains(2)
                     && NoValidos.contains(3) && NoValidos.contains(4)
                     && NoValidos.contains(5) && NoValidos.contains(6)
                     && NoValidos.contains(7) && NoValidos.contains(8)
                     && NoValidos.contains(9)) {

                 hecho = false;
                 return hecho;
             }

             for (int k=1; k <= 9; k++) {

                 if (!NoValidos.contains(k)) {

                     tablaRest.setValor(k);
                     tabla[tablaRest.getFila()][tablaRest.getColumna()] = tablaRest;

                     EncontrarInvalidos();
```

```
                    if( funciona() == true) {
                        return true;
                    } else {
                        tablaRest.setValor(0);
                    }
                }
            }

            return hecho;

}

public void printTablero() {
    System.out.println();
            for (int i = 0; i < 9; i++) {
                if ( i == 3 || i == 6 || i==9 ) {
                    System.out.println("                                        ");
                }
                for (int j=0; j <9; j++) {
                    if( j==2 || j==5 || j==8) {
                        System.out.print(tabla[i][j].getValor()+ " || "); // Con esto "dibujamos" las cajas, cada
                    } else {
                        System.out.print(tabla[i][j].getValor()+ " | ");
                    }
                }
                System.out.println();
            }
}
```

Una vez almacenados todos los valores no válidos para cada cuadrícula. Pasamos al método funciona, el cual establecerá como restricciones todos los valores inválidos e irá probando diferentes valores. Siempre que el valor que coloque, no cumpla todas las restricciones, lo añadirá como no válido y colocará el valor a 0 nuevamente hasta encontrar un camino o solución válida.

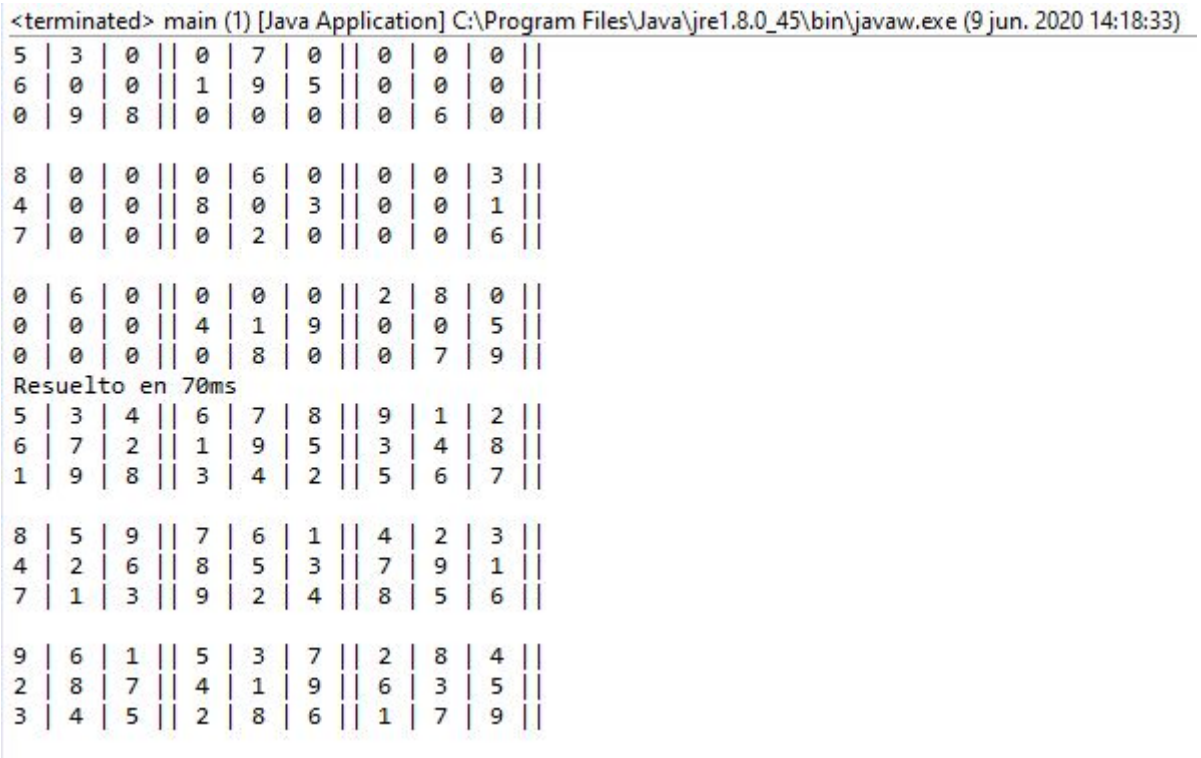
Por último, necesitamos la clase Main o iniciadora para la resolución del problema.

```java
public class main {     // Este va a ser nuestro main


    // Creamos una variable para el tiempo, que sea estatica
    // 3 variables, el tiempo inicial, final y total

    static long Tini=0, Tfinal=0, Ttotal=0;

    public static void main(String[] args) {

        //Mediante array pasaremos el sudoku a resolver al programa

        ArrayList<Integer> SudokuMapa = new ArrayList <Integer>();
        Collections.addAll(SudokuMapa,  // El segundo parametro sera el mapa de 9x9
                5,3,0,0,7,0,0,0,0,
                6,0,0,1,9,5,0,0,0,
                0,9,8,0,0,0,0,6,0,
                8,0,0,0,6,0,0,0,3,
                4,0,0,8,0,3,0,0,1,
                7,0,0,0,2,0,0,0,6,
                0,6,0,0,0,0,2,8,0,
                0,0,0,4,1,9,0,0,5,
                0,0,0,0,8,0,0,7,9);  // Sudoku ejemplo sacado de google imagenes

        Tablero su= new Tablero(SudokuMapa);
        IA resuelve = new IA(su.Pasartabla()); // Le pasamos el estado inicial del tablero a la IA
        resuelve.EncontrarInvalidos();
        Tini=System.currentTimeMillis();

        //Ahora necesitariamos saber si es posible o no resolver el sudoku

        if(resuelve.funciona()) {

            Tfinal=System.currentTimeMillis();
            Ttotal = Tfinal-Tini;
            System.out.println("Resuelto en " +Ttotal+ "ms");
            //Imprimimos la solucion del sudoku
            su.printTablero();

        } else { // Los demas casos no tendran solucion

            System.out.println("No he encontrado una solucion ");

        }
```

Dentro de esta clase, estableceremos el sudoku a resolver. Como añadido, he introducido las variables de tiempo para saber cuánto ha tardado en encontrar la solución.

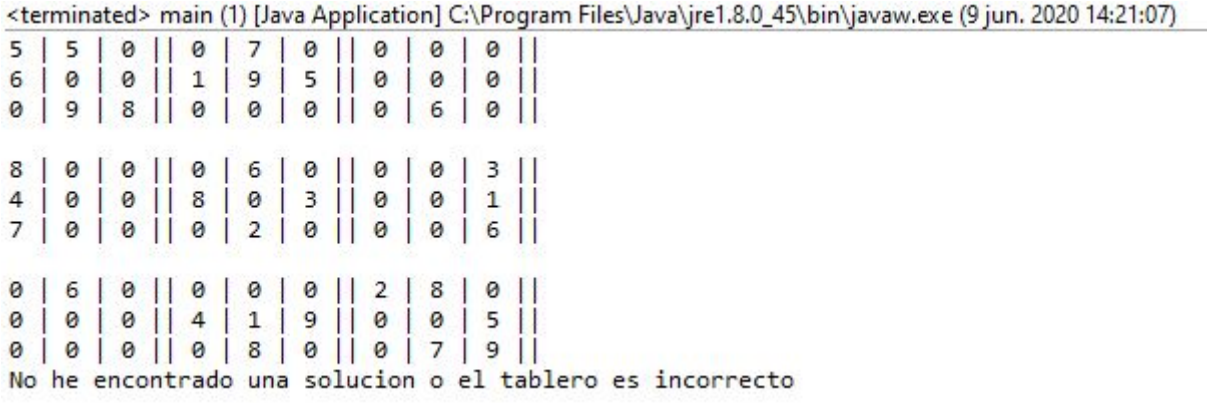
Pruebas

Para realizar las pruebas, hemos utilizado un sudoku que sabemos que tiene solución y otro que no. En el primer caso la respuesta obtenida por consola ha sido la siguiente.



Resolviendo correctamente el sudoku. No colocaremos más capturas de este tipo, ya que todos las pruebas con sudokus correctos daban un resultado similar. Tiempo similar y cuadrículas correctas.

Cuando colocamos un sudoku o tablero incorrecto, la salida por consola es la siguiente.



Por lo que en un principio podemos concluir que nuestra IA funciona correctamente.

4. Conclusiones

De todo lo anterior se puede concluir que, el algoritmo cumple a la perfección su cometido. No obstante es susceptible a mejoras, tanto en el código de la IA como en el código del programa general.

Como posibles mejores podríamos tener el incluir una interfaz gráfica realizada con un JFrame, que nos permita introducir el sudoku manualmente sin necesidad de tocar el código. Una reducción del código de la IA o reinterpretación ya que la cantidad de bucles realizados pueden llevar a confusión y algunos de ellos podrían simplificarse o reinterpretarse de una manera más visual, aunque probablemente deberíamos también cambiar el método de búsqueda de las soluciones.

5. Valoración personal

En general, estoy muy satisfecho del trabajo realizado en estas últimas 48h. Con este trabajo opcional intento demostrar mis conocimientos del primer bloque de la materia ya que creo que la calificación obtenida en ese primer bloque no representa los conocimientos adquiridos. Por eso mismo, decidí realizar esta práctica sobre Backtracking y búsqueda con restricciones.

La parte del desarrollo fue sin duda la más complicada. Desarrollar y pensar cómo quería que funcionase el algoritmo sin llegar a copiar el trabajo de otros.

Probé de varias maneras la implementación de la IA y al final, el utilizar las coordenadas como falsos “nodos” mediante matrices y arraylist fue lo mejor. Ya que intenté en una primera instancia realizar un JFrame, definir unos cuadrados y colocar como nodo cada cuadrado dentro de la ventana pero por falta de experiencia con interfaces gráficas y el uso de estas en métodos decidí realizarlo de esta otra forma más rudimentaria pero funcional.

6. Bibliografía

- https://cursos.com/sudoku/
- https://informatica.cv.uma.es/pluginfile.php/357004/mod_resource/content/2/Sist.%20Inteligentes%20Busqueda.pdf
- https://www.youtube.com/watch?v=tNe48XHLXVw