

Práctica Evaluable 4

Análisis y Diseño de Algoritmos

Nombre: Emilio Gómez Esteban

Grupo: 2ºD (Grado Matemáticas + Ingeniería Informática)

CÓDIGO EMPLEADO (CLASE TABLEROSUDOKU):

```
import java.util.*;

public class TableroSudoku implements Cloneable {

    // constantes relativas al nº de filas y columnas del tablero
    protected static final int MAXVALOR=9;
    protected static final int FILAS=9;
    protected static final int COLUMNAS=9;

    protected static Random r = new Random();

    protected int celdas[][]; // una celda vale cero si está libre.

    public TableroSudoku() {
        celdas = new int[FILAS][COLUMNAS]; //todas a cero.
    }

    // crea una copia de su parámetro
    public TableroSudoku(TableroSudoku uno) {
        TableroSudoku otro = (TableroSudoku) uno.clone();
        this.celdas = otro.celdas;
    }

    // crear un tablero a partir de una configuración inicial (las celdas
    vacías // se representan con el caracter ".".
    public TableroSudoku(String s) {
        this();
        if(s.length() != FILAS*COLUMNAS) {
            throw new RuntimeException("Construcci\u00D3n de sudoku no
v\u00E9lida.");
        } else {
            for(int f=0;f<FILAS;f++)
                for(int c=0;c<COLUMNAS;c++) {
                    Character ch = s.charAt(f*FILAS+c);
                    celdas[f][c] = (Character.isDigit(ch) ?
Integer.parseInt(ch.toString()) : 0 );
                }
        }
    }

    /* Realizar una copia en profundidad del objeto
```

```

    * @see java.lang.Object#clone()
    */
    public Object clone() {
        TableroSudoku clon;
        try {
            clon = (TableroSudoku) super.clone();
            clon.celdas = new int[FILAS][COLUMNAS];
            for(int i=0; i<celdas.length; i++)
                System.arraycopy(celdas[i], 0, clon.celdas[i], 0,
celdas[i].length);
        } catch (CloneNotSupportedException e) {
            clon = null;
        }
        return clon;
    }

    /* Igualdad para la clase
    * @see java.lang.Object#equals()
    */
    public boolean equals(Object obj) {
        if (obj instanceof TableroSudoku) {
            TableroSudoku otro = (TableroSudoku) obj;
            for(int f=0; f<FILAS; f++)
                if(!Arrays.equals(this.celdas[f], otro.celdas[f]))
                    return false;
            return true;
        } else
            return false;
    }

    public String toString() {
        String s = "";

        for(int f=0; f<FILAS; f++) {
            for(int c=0; c<COLUMNAS; c++)
                s += (celdas[f][c]==0 ? "." :
String.format("%d", celdas[f][c]));
            return s;
        }
    }

    // devuelve true si la celda del tablero dada por fila y columna está
vacía.
    protected boolean estaLibre(int fila, int columna) {
        return celdas[fila][columna] == 0;
    }

    // devuelve el número de casillas libres en un sudoku.
    protected int numeroDeLibres() {
        int n=0;
        for (int f = 0; f < FILAS; f++)
            for (int c = 0; c < COLUMNAS; c++)
                if(estaLibre(f,c))
                    n++;
        return n;
    }

```

```

protected int numeroDeFijos() {
    return FILAS*COLUMNAS - numeroDeLibres();
}

// Devuelve true si @valor ya esta en la fila @fila.
protected boolean estaEnFila(int fila, int valor) {
    // A completar por el alumno

    boolean encontrado=false;
    int j=0;

    while(j<COLUMNAS && !encontrado) {
        if(valor == celdas[fila][j]) {
            encontrado = true;
        }
        j++;
    }

    return encontrado;
}

// Devuelve true si @valor ya esta en la columna @columna.
protected boolean estaEnColumna(int columna, int valor) {
    // A completar por el alumno

    boolean encontrado=false;
    int i=0;

    while(i<FILAS && !encontrado) {
        if(valor == celdas[i][columna]) {
            encontrado = true;
        }
        i++;
    }

    return encontrado;
}

// Devuelve true si @valor ya esta en subtablero al que pertenece @fila
y @columna.
protected boolean estaEnSubtablero(int fila, int columna, int valor) {
    // A completar por el alumno

    boolean encontrado = false;
    int i = fila - (fila%3);
    int j = columna - (columna%3);

    int colInicio = j;
    int maxj = j+3;
    int maxi = i+3;

    while(i < maxi && !encontrado) {
        j=colInicio;

        while(j < maxj && !encontrado) {
            if(celdas[i][j] == valor) {
                encontrado=true;
            }
        }
        i++;
    }
}

```

```

        }
        j++;
    }
    i++;
}

return encontrado;
}

// Devuelve true si se puede colocar el @valor en la @fila y @columna
dadas.
protected boolean sePuedePonerEn(int fila, int columna, int valor) {
    // A completar por el alumno
    return (!estaEnFila(fila,valor)) && (!estaEnColumna(columna,
valor))
        && (!estaEnSubtablero(fila,columna,valor));
}

protected void resolverTodos(List<TableroSudoku> soluciones, int fila,
int columna) {
    // A completar por el alumno

    if(numeroDeLibres() == 0) {
        soluciones.add(new TableroSudoku(this));
    } else {
        if(estaLibre(fila, columna)) {

            for(int candidato=1; candidato<=9; candidato++) {
                if(sePuedePonerEn(fila, columna, candidato))

                    {
                        celdas[fila][columna] = candidato;
                        int j;
                        int i;
                        if(columna == 8) {
                            j=0;
                            i=fila+1;
                        } else {
                            i = fila;
                            j = columna +1;
                        }
                        resolverTodos(soluciones,i,j);
                        celdas[fila][columna]=0;
                    }
                }
            } else {
                if(columna==8) {
                    columna=0;
                    fila++;
                } else {
                    columna++;
                }
                resolverTodos(soluciones, fila, columna);
            }
        }
    }
}

```

```

    public List<TableroSudoku> resolverTodos() {
        List<TableroSudoku> sols = new LinkedList<TableroSudoku>();
        resolverTodos(sols, 0, 0);
        return sols;
    }

    public static void main(String arg[]) {
        TableroSudoku t = new TableroSudoku(
            ".4....36263.941...5.7.3.....9.3751..3.48.....17..62...716.9..2...96.....31
            2..9.");
        List<TableroSudoku> lt = t.resolverTodos();
        System.out.println(t);
        System.out.println(lt.size());
        for(Iterator<TableroSudoku> i= lt.iterator(); i.hasNext();) {
            TableroSudoku ts = i.next();
            System.out.println(ts);
        }
    }
}

```

ESTRATEGIA EMPLEADA:

Definición del problema: En un sudoku disponemos de un tablero de tamaño 9x9 en cuyas celdas se pueden situar valores entre 1 y 9. A su vez, el tablero queda dividido en 9 subtableros de tamaño 3x3. El valor de algunas celdas está fijado inicialmente. El juego consiste en completar los valores de las demás celdas de modo que se cumplan las siguientes reglas:

- 1) Un mismo valor no puede aparecer más de una vez en la misma fila del tablero,
- 2) Un mismo valor no puede aparecer más de una vez en la misma columna del tablero,
- 3) Y un mismo valor no puede aparecer más de una vez en el mismo subtablero.

Aunque normalmente se usan configuraciones iniciales que solo permiten una única solución (son lo que se llaman *sudokus bien formados*), en esta práctica consideraremos sudokus que puedan admitir varias soluciones correctas. Por tanto, el objetivo será implementar un algoritmo, mediante la técnica de vuelta atrás, que obtenga todas las soluciones posibles para una configuración inicial de sudoku.

En primer lugar, implementaremos los siguientes métodos auxiliares:

- Método *estaEnFila(int fila, int valor)*: nos devolverá *True* si el valor comparado ya aparece en alguna posición de la fila dada.
- Método *estaEnColumna(int columna, int valor)*: nos devolverá *True* si el valor comparado ya aparece en alguna posición de la columna dada.
- Método *estaEnSubtablero(int fila, int columna, int valor)*: nos devolverá *True* si el valor comparado ya aparece en alguna de la celda del subtablero al que corresponde la posición dada por la fila y por la columna.
- Método *sePuedePonerEn(int fila, int columna, int valor)*: nos devolverá *True* si el valor comparado no está ni en la fila dada, ni en la columna dada, ni en el subtablero al que corresponde la posición dada por la fila y por la columna.

A continuación, usando estos métodos y otros ya implementados en la clase *TableroSudoku* se trata de completar el método *resolverTodos(List<TableroSudoku> soluciones, int fila, int columna)* el cual resolverá el problema mediante la técnica de vuelta atrás. Para ello, añadirá a la lista soluciones todas las soluciones válidas para el tablero que se pueden obtener rellenando las celdas a partir de la fila y la columna.

Solución vuelta atrás:

- Estructura de la solución: lista de todas las soluciones válidas para el tablero inicial dado.
- Estado inicial: lista vacía.
- Política de ramificación:
 - Restricciones explícitas: los valores que se colocarán en cada celda estarán comprendidos entre 1 y 9 (ambos incluidos).
 - Restricciones implícitas: el valor que se trata de colocar en la celda, no puede estar repetido en su fila ni en su columna ni en el subtablero al que pertenece su posición.
- Función de terminación: un tablero será solución del sudoku cuando no queden celdas libres en él.
- Función objetivo: no es necesario, no se pide optimizar la resolución.

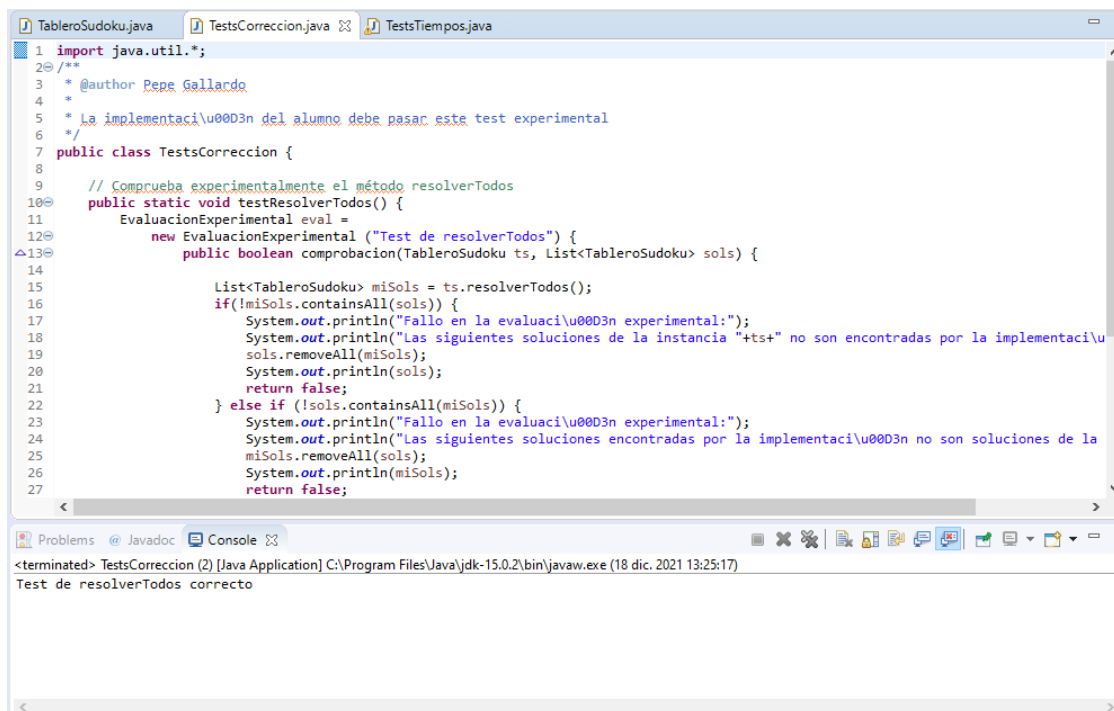
El caso base de la recursividad será aquel en que no quedan celdas libres, es decir, el *numeroDeLibres* es igual a 0, y en ese caso, se añade el tablero completo a la lista de soluciones (lo correcto en este caso es introducir una copia del objeto *this* en la lista cada

vez que se encuentre una solución, ya que si modificamos este objeto posteriormente también se modificará el introducido en la lista).

En otro caso, vemos si la celda estudiada está libre (si está ocupada se salta a la siguiente y llamamos recursivamente a la función). En ese caso, comprobamos si cada uno de los candidatos se puede colocar en esa celda (del 1 al 9). Primero lo colocamos y a continuación, llamamos recursivamente a nuestra función con la siguiente celda (si llegamos al final de una fila, tomamos el elemento primero de la siguiente fila). Ahora bien, en caso de que nuestra hipotética solución no sea correcta, nuestro programa realizará la vuelta atrás (gracias a la recursividad) e irá colocando todas las celdas completadas (erróneamente) a 0, es decir, libres.

EVALUACIÓN EXPERIMENTAL DE LA IMPLEMENTACIÓN:

Tomando la clase TestsCorreccion, se puede realizar una comprobación del algoritmo usado. En este caso, la respuesta conseguida es la siguiente:



```
1 import java.util.*;
2 /**
3  * @author Pepe Gallardo
4  *
5  * La implementación del alumno debe pasar este test experimental
6  */
7 public class TestsCorreccion {
8
9     // Comprueba experimentalmente el método resolverTodos
10    public static void testResolverTodos() {
11        EvaluacionExperimental eval =
12            new EvaluacionExperimental("Test de resolverTodos") {
13            public boolean comprobacion(TableroSudoku ts, List<TableroSudoku> sols) {
14
15                List<TableroSudoku> miSols = ts.resolverTodos();
16                if(!miSols.containsAll(sols)) {
17                    System.out.println("Fallo en la evaluación experimental:");
18                    System.out.println("Las siguientes soluciones de la instancia "+ts+" no son encontradas por la implementación");
19                    sols.removeAll(miSols);
20                    System.out.println(sols);
21                    return false;
22                } else if (!sols.containsAll(miSols)) {
23                    System.out.println("Fallo en la evaluación experimental:");
24                    System.out.println("Las siguientes soluciones encontradas por la implementación no son soluciones de la instancia "+ts+"");
25                    miSols.removeAll(sols);
26                    System.out.println(miSols);
27                    return false;
28                }
29            }
30        };
31    }
32}
```

Problems Javadoc Console

<terminated> TestsCorreccion (2) [Java Application] C:\Program Files\Java\jdk-15.0.2\bin\javaw.exe (18 dic. 2021 13:25:17)

Test de resolverTodos correcto

Con las clases *Temporizador* y *Gráfica* suministradas, se podrán realizar gráficas para ver experimentalmente el comportamiento de la implementación, las cuales muestran el tiempo medio para resolver sudokus con distinto número de celdas fijadas inicialmente. Para ello, se ejecuta la clase *TestTiempos* y se obtiene el siguiente resultado:

