Práctica 4. Sudokus. Vuelta Atrás.

María Peinado Toledo. Doble Grado Ingeniería Informática y Matemáticas.

Vamos a tratar de resolver un Sudoku a través de aplicar el método de la Vuelta Atrás. Tenemos un tablero 9x9 y se aplican las reglas ya conocidas para resolverlo.

Veamos la clase TableroSudoku.java utilizada para resolver el problema:

```
// ALUMNO: María Peinado Toledo
// GRUPO: D. Doble Grado Ingeniería Informática y Matemáticas

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 est\u00E1 libre.

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

    // crea una copia de su par\u00E1metro
    public TableroSudoku(TableroSudoku uno) {
        TableroSudoku otro = (TableroSudoku) uno.clone();
        this.celdas = otro.celdas;
    }

 // crear un tablero a parir de una configuraci\u00D3n inicial (las celdas vac\u00EDas
 // se representan con el caracter ".".
 public TableroSudoku(String s) {
     this();
     if(s.length() != FILAS*COLUMNAS) {
         throw new RuntimeException("Construcci\u00D3n de sudoku no v\u00E1lida.");
     } 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;
 }
```

```java
/* 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;
}


// devuelva true si la celda del tablero dada por fila y columna est\u00E1 vac\u00EDa.
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) {
    int i=0;
    boolean encontrado=false;
    while(!encontrado&&i<9) {
        if(celdas[fila][i]==valor) {
            encontrado=true;
        }
        i++;
    }
    return encontrado;
}
```

```java
// Devuelve true si @valor ya esta en la columna @columna.
protected boolean estaEnColumna(int columna, int valor) {
    int i=0;
    boolean encontrado=false;
    while(!encontrado&&i<9) {
        if(celdas[i][columna]==valor) {
            encontrado=true;
        }
        i++;
    }
    return encontrado;
}


// Devuelve true si @valor ya esta en subtablero al que pertence @fila y @columna.
protected boolean estaEnSubtablero(int fila, int columna, int valor) {
    int fi=fila-(fila%3),co=columna-(columna%3);
    boolean encontrado=false;
    int aux=co;
    int maxC=co+3, maxF=fi+3;
    while(!encontrado&&fi<maxF) {
        co=aux;
        while(!encontrado&&co<maxC) {
            if(celdas[fi][co]==valor) {
                encontrado=true;
            }
            co++;
        }
        fi++;
    }
    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) {
    return (!estaEnFila(fila,valor)) && (!estaEnColumna(columna,valor)) && (!estaEnSubtablero(fila,columna,valor));
}
```

Estudiemos el algoritmo utilizado:

```java
protected void resolverTodos(List<TableroSudoku> soluciones, int fila, int columna) {
    int co,fi;
    if(numeroDeLibres()==0) {
        soluciones.add(new TableroSudoku(this));
    }else{
        if(estaLibre(fila,columna)) {
            for(int num=1;num<=9;num++) {
                if(sePuedePonerEn(fila,columna,num)) {
                    celdas[fila][columna]=num;
                    if(columna==8) {
                        co=0;
                        fi=fila+1;
                    }else {
                        fi=fila;
                        co=columna+1;
                    }
                    resolverTodos(soluciones,fi,co);
                    celdas[fila][columna]=0;
                }
            }
        }else {
            if(columna==8) {
                columna=0;
                fila++;
            }else {
                columna++;
            }
            resolverTodos(soluciones,fila,columna);
        }
    }
}
```

Para empezar, comprobamos que sigan quedando espacios libres por rellenar. En el caso de que no, habríamos terminado una solución y la añadimos a nuestra lista de soluciones.
En el caso de que queden, vemos si la fila y la columna con la que estamos tratando está libre, en el caso de que no, avanzamos por el tablero. Si estuviese libre, con un for vamos creando las posibles soluciones tras añadir un número del 1 al 9 de la siguiente manera. Vemos si ese número cumple las reglas del sudoku, en el caso de que sí, ocupamos esa celda con ese número y avanzamos de celda y llamamos recursivamente al algoritmo. Después de llamar al algoritmo de nuevo, una vez haya completado el sudoku o no encuentre solución con los números ya introducidos, retrocediendo con la recursividad, vamos a ir vaciando de nuevo el sudoku para una nueva solución.

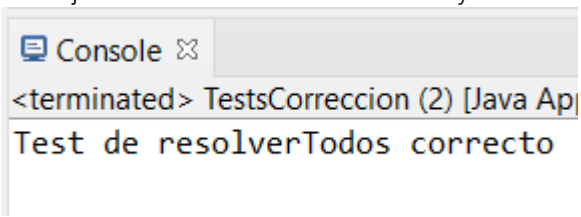
```
public List<TableroSudoku> resolverTodos() {
    List<TableroSudoku> sols  = new LinkedList<TableroSudoku>();
    resolverTodos(sols, 0, 0);
    return sols;
}
```

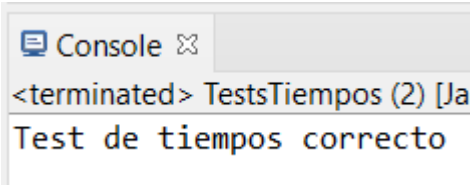
```
public static void main(String arg[]) {
    TableroSudoku t = new TableroSudoku(
            ".4....36263.941...5.7.3.....9.3751..3.48.....17..62...716.9..2...96.......312..9.");
    List<TableroSudoku> lt = t.resolverTodos();
    System.out.println(t);
    System.out.println(lt.size());
    for(Iterator<TableroSudoku> i= lt.iterator(); i.hasNext();) {
        TableroSudoku ts = i.next();
        System.out.println(ts);

    }

}
```

Tras ejecutar las clases TestsCorrecion y TestTiempos se muestra lo siguiente por pantalla:





Adjunto las gráficas resultado de la práctica: