Juan Carlos Alcausa Luque

Memoria Práctica 4: ADA

1. Descripción 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 (indicados con distintos colores en las figuras posteriores). 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.

2. Código implementado

```
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;
      }

      /* 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) {
		boolean esta = false;
		int i=0;
		while(i<FILAS && !esta) {
			if(celdas[fila][i]==valor) {
				esta=true;
			}
			i++;
		}
		return esta;
	}

	// Devuelve true si @valor ya esta en la columna @columna.
	protected boolean estaEnColumna(int columna, int valor) {
```

```
            boolean esta = false;
            int i=0;
            while(i<COLUMNAS && !esta) {
                  if(celdas[i][columna]==valor) {
                        esta=true;
                  }
                  i++;
            }
            return esta;
      }


      // Devuelve true si @valor ya esta en subtablero al que
pertence @fila y @columna.
      protected boolean estaEnSubtablero(int fila, int columna,
int valor) {
            boolean esta = false;
            int f= fila - (fila%3);
            int c = columna - (columna%3);
            int aux=c;
            int maxFil = f+3;
            int maxCol = c+3;
            while(f<maxFil && !esta) {
                  c=aux;
                  while(c<maxCol && !esta) {
                        if(celdas[f][c]==valor) {
                              esta=true;
                        }
                        c++;
                  }
                  f++;
            }
            return esta;
      }


      // 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));
      }

      private int [] proxPos(int fila, int columna) {
            int fils = fila;
            int cols = columna;
```

```
            while(fils< FILAS) {
                  if(estaLibre(fils, cols)) {
                        int [] aux = new int [2];
                        aux[0]= fils;
                        aux[1]=cols;
                        return aux;
                  }
                  cols++;
                  if(cols==COLUMNAS) {
                        cols=0;
                        fils++;
                  }
            }
            return null;
      }

      protected void resolverTodos(List<TableroSudoku>
soluciones, int fila, int columna) {
            int [] prox = proxPos(fila, columna);

            if(prox==null) {
                  soluciones.add(new TableroSudoku(this));
            }else {
                  int fil= prox[0];
                  int col= prox[1];

                  for (int i=1; i<=MAXVALOR; i++) {
                        if(sePuedePonerEn(fil, col, i)) {
                              celdas[fil][col]=i;
                              resolverTodos(soluciones,fil,col);
                        }
                  }

                  celdas[fil][col]=0;
            }
      }


      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);

            }

      }


}
```

3. Decisiones de diseño

Para resolver el problema, utilizamos la técnica vuelta atrás, que se consigue con el método resolverTodos(List<TableroSudoku> soluciones, int fila, int columna) que añade a la lista las soluciones válidas que se obtienen rellenando las celdas a partir de la fila y la columna inclusive. Antes de realizar este método debemos implementar otros métodos auxiliares.

En primer lugar para el método *estaEnFila*, creamos una variable boolean auxiliar que será la que devolveremos con el resultado. A continuación recorremos todas las celdas de esa fila, dejando fija esta e iterando sobre las columnas, en caso de que la encontremos antes saldremos del bucle while.

El método *estaEnColumna*, es exactamente igual al anterior pero en vez de iterar sobre las columnas dejando fijas la fila, dejamos fija la columna e iteramos sobre las filas.

El siguiente método *estaEnSubtablero*, devuelve un boolean indicando si el valor dado pertenece al subtablero indicado por la posición dado. El tablero se divide en nueve tableros de 3x3, por lo que debemos localizar a cual pertenece y luego localizar si el valor indicado se encuentra en esta.

Por último, para implementar el algoritmo de vuelta atrás, es decir el método *resolverTodos*, antes creamos el método privado proxPos, que nos genera los posibles próximos candidatos para nuestro algoritmo de vuelta atrás en caso de que los haya.

Utilizamos esta clase auxiliar para decidir si podemos añadir o no más valores, es decir, si está completa ya la tabla. En caso de no estarlo tomamos los parámetros de las fila y columna de la siguiente posición libre e intentamos añadir un valor que cumpla los requisitos indicados iterando en el bucle for. Si lo añade, cambiamos la tabla añadiendo el valor en la posición y llamamos de nuevo a la función con el nuevo estado de la tabla añadiendo finalmente que en caso de necesitar hacer vuelta atrás dejemos la posición como vacía.